Solaris 模块调试器指南



Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

文件号码 819–7055–10
2006 年 11 月

061127@15490

目录

前言

模块调试器 (Modular Debugger, MDB) 是用于 Solaris™ 操作系统的高度可扩展的通用调试工具。《Solaris 模块调试器指南》介绍如何使用 MDB 调试复杂的软件系统，尤其着重介绍了可用于调试 Solaris 内核以及关联的设备驱动程序和模块的工具。 本书还包括对 MDB 语言语法、调试器功能和 MDB 模块编程 API 的完整参考和讨论。

注 – 此 Solaris 发行版支持使用以下 SPARC® 和 x86 系列处理器体系结构的系统：UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T。支持的系统可以在 `http://www.sun.com/bigadmin/hcl` 上的《Solaris 10 Hardware Compatibility List》中找到。本文档列举了在不同类型的平台上进行实现时的所有差别。

在本文档中，术语 "x86" 指使用与 AMD64 或 Intel Xeon/Pentium 产品系列兼容的处理器生产的 64 位和 32 位系统。若想了解本发行版支持哪些系统，请参见《Solaris 10 Hardware Compatibility List》。

目标读者

如果您是一名侦探，正在调查犯罪现场，您可能会采访目击者，并要求他们描述发生的情况和所看到的人员。 但是，如果没有目击者或者这些描述证明不够充分，您可能考虑收集指纹和法庭证据，以便进行 DNA 检查来帮助破案。通常，软件程序失败分成类似的两类：可以用源代码级调试工具解决的问题，以及需要底层调试工具、检查核心文件和了解汇编语言才能诊断和更正的问题。 MDB 是专门用于简化对这第二类问题进行分析的调试器。

也许不需要在每种情况下都使用 MDB，就像侦探不需要显微镜和 DNA 证据来侦破每个犯罪行为。但是，为诸如操作系统之类的复杂底层软件系统编程时，这些情况可能会频繁出现。 因此，MDB 设计为一个调试框架，可以使用它构造自己的自定义分析工具，从而帮助诊断这些问题。 MDB 还提供了一组功能强大的内置命令，可以使用它们在汇编语言级别上分析程序的状态。

如果您不熟悉汇编语言编程和调试，请参见第 10 页中的 "相关书籍和文章"，您可能会发现其中提供的参考资料是很有用的。

您还应该反汇编将要调试的程序中相关的各种函数，以便熟悉程序的源代码与对应汇编语言代码之间的关系。 如果计划使用 MDB 调试 Solaris 内核软件，则应该仔细阅读第 8 章和第 9 章。这两章提供有关为调试 Solaris 内核软件而提供的 MDB 命令和工具的更详细信息。

本书的结构

第 1 章概述调试器。本章适用于所有用户。

第 2 章介绍 MDB 体系结构，并说明在整本书中使用的调试器概念的术语。 本章适用于所有用户。

第 3 章介绍 MDB 语言的语法、运算符和计算规则。 本章适用于所有用户。

第 4 章介绍 MDB 交互式命令行编辑工具和输出页面调度程序。 本章适用于所有用户。

第 5 章介绍一组始终可用的内置调试器命令。 本章适用于所有用户。

第 6 章介绍用于控制实时运行程序的执行的 MDB 工具。本章适用于应用程序开发者和设备驱动程序开发者。执行控制功能对于系统管理员可能也是很有用的。

第 7 章介绍用于控制特定于 kmdb 的实时操作系统内核的执行的 MDB 工具。本章适用于操作系统内核开发者和设备驱动程序开发者。

第 8 章介绍一组为调试 Solaris 内核而提供的可装入调试器命令。 本章适用于打算检查 Solaris 内核崩溃转储的用户和内核软件开发者。

第 9 章介绍 Solaris 内核内存分配器的调试功能和为利用这些功能而提供的 MDB 命令。 本章适用于高级程序员和内核软件开发者。

第 10 章介绍用于写入可装入调试器模块的工具。 本章适用于高级程序员和打算开发对 MDB 的自定义调试支持的软件开发者。

附录 A 提供 MDB 命令行选项的参考。

附录 B 提供有关使用调试器的警告和说明。

附录 C 提供 adb 命令及其 MDB 等效项的参考。adb 命令由 mdb 实现。

附录 D 提供 crash 命令及其 MDB 等效项的参考。Solaris 中不再存在 crash 命令。

相关书籍和文章

建议您阅读以下与需要执行的任务有关的书籍和文章：

- 由 Vahalia, Uresh 编著的《UNIX Internals: The New Frontiers》。Prentice Hall 出版，1996。ISBN 0-13-101908-2
- 由 Mauro, Jim 和 McDougall, Richard 合著的《Solaris Internals: Core Kernel Components》。Sun Microsystems Press 出版，2001。ISBN 0-13-022496-0
- 《The SPARC Architecture Manual, Version 9》。Prentice Hall 出版，1998。ISBN 0–13–099227–5
- 《The SPARC Architecture Manual, Version 8》。Prentice Hall 出版，1994。ISBN 0-13-825001-4

- 《Pentium Pro Family Developer's Manual, Volumes 1-3》。Intel Corporation 出版，1996。ISBN 1-55512-259-0（第 1 卷），ISBN 1-55512-260-4（第 2 卷），ISBN 1-55512-261-2（第 3 卷）
- 由 Bonwick, Jeff 编著的《The Slab Allocator:An Object-Caching Kernel Memory Allocator》。Usenix 会议 1994 年文献汇编。ISBN 9–99–452010–5
- 《SPARC Assembly Language Reference Manual》。Sun Microsystems 出版，1998。
- 《x86 Assembly Language Reference Manual》。Sun Microsystems 出版，1998。
- 《Writing Device Drivers》。Sun Microsystems 出版，2000。
- 《STREAMS Programming Guide》。Sun Microsystems 出版，2000。
- 《 Solaris 64-bit Developer's Guide》。Sun Microsystems 出版，2000。
- 《Linker and Libraries Guide》。Sun Microsystems 出版，2000。

联机访问 Sun 文档

可以通过 docs.sun.com[SM] Web 站点联机访问 Sun 技术文档。您可以浏览 docs.sun.com 文档库或查找某个特定的书名或主题。URL 为 `http://docs.sun.com`。

印刷约定的含义

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体或符号	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `machine_name% you have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同	`machine_name%` **`su`** `Password:`
AaBbCc123	要使用实名或值替换的命令行占位符	要删除文件，请键入 **`rm`** *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *class* 选项。
新词术语强调	新词或术语以及要强调的词	请**勿**保存文件。 必须成为**超级用户**才能执行此操作。

表 P–1 印刷约定 （续）

字体或符号	含义	示例
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表列出了 C shell、Bourne shell 和 Korn shell 的缺省系统提示符和超级用户提示符，以及 MDB 调试器提示符。

表 P–2 Shell 提示符

Shell	提示符
C shell	`machine_name%`
C shell 超级用户	`machine_name#`
Bourne shell 和 Korn shell	`$`
Bourne shell 和 Korn shell 超级用户	`#`
`mdb`	`>`
`kmdb`	`[`*cpu*`]>`

第 1 章

模块调试器概述

模块调试器 (Modular Debugger, MDB) 是用于 Solaris 的通用调试工具，其主要特性是可扩展性。 本书介绍如何使用 MDB 调试复杂的软件系统，尤其着重介绍了可用于调试 Solaris 内核以及关联的设备驱动程序和模块的工具。 本书还提供了 MDB 语言语法、调试器功能和 MDB 模块编程 API 的完整参考并对它们进行了讨论。

简介

调试是分析软件程序的执行过程和状态以便去除缺陷的过程。 传统的调试工具提供了执行控制功能，以便程序员可以在受控环境中重新执行程序，并显示程序数据的当前状态或以用于开发程序的源语言计算表达式。遗憾的是，这些方法通常不适合用于调试复杂的软件系统，例如：

- 可能无法重现错误且具有大量相对分散的程序状态的操作系统
- 已高度优化的程序或已删除其调试信息的程序
- 本身就是底层调试工具的程序
- 开发者只能访问事后信息的客户情况

MDB 是一种工具，它可为调试这些程序和方案提供完全可自定义环境，其中包括程序员用来实现其自己的调试命令以执行程序特定的分析的动态模块工具。每个 MDB 模块都可用于在几种不同的上下文（包括实时的和事后的）中检查程序。Solaris 操作系统包含一组 MDB 模块，旨在帮助程序员调试 Solaris 内核以及相关的设备驱动程序和内核模块。第三方开发者可能会发现，为监控程序或用户软件开发和提供自定义的调试模块是很有用的。

MDB 功能

MDB 可提供范围广泛的功能集合，用于分析 Solaris 内核和其他目标程序。您可以：

- 执行 Solaris 内核崩溃转储和用户进程核心转储的事后分析：除标准数据显示和格式设置功能外，MDB 还包含用于简化复杂的内核和进程状态分析过程的调试器模块集合。 使用调试器模块可以实现复杂查询的公式化，以便：

  - 查找特定线程分配的所有内存
  - 列显内核 STREAM 的直观图
  - 确定特定地址所引用的结构类型
  - 在内核中查找已泄漏的内存块
  - 分析内存以查找栈跟踪
- 使用高级编程 API 实现自己的调试器命令和分析工具，而不必重新编译或修改调试器本身：在 MDB 中，调试支持的实现形式为一组可装入模块（调试器可以 dlopen(3C) 的共享库），其中每个可装入模块都提供一组扩展调试器本身功能的命令。相应地，调试器提供核心服务的 API，例如读写内存和访问符号表信息的功能。MDB 为开发者提供一个用于针对其自身的驱动程序和模块实现调试支持的框架；然后使这些模块可供所有人使用。
- 了解如何使用 MDB（如果您已熟悉传统的调试工具 adb 和 crash）： MDB 为这些现有调试解决方案提供向后兼容性。MDB 语言本身是 adb 语言的超集；所有现有 adb 宏和命令都在 MDB 内工作，因此使用 adb 的开发者可以立即使用 MDB，而无需了解任何 MDB 特定的命令。 MDB 还提供了功能比 crash 实用程序更强大的命令。
- 增强的可用性功能可带来许多益处。 MDB 提供了许多可用性功能，其中包括：
  - 命令行编辑
  - 命令历史记录
  - 内置输出页面调度程序
  - 语法错误检查和处理
  - 联机帮助
  - 交互式会话日志记录

使用 MDB

在 Solaris 系统中，以共享通用功能的两个命令提供 MDB：mdb 和 kmdb。可以使用 mdb 命令以交互方式或在脚本中调试实时用户进程、用户进程核心转储文件、内核崩溃转储、实时操作系统、目标文件和其他文件。在还需要控制和停止内核执行时， 可以使用 kmdb 命令调试实时操作系统内核和设备驱动程序。要启动 mdb， 请执行 mdb(1) 命令。要启动 kmdb， 请引导系统（如 kmdb(1) 手册页中所述）或执行带 -K 选项的 mdb 命令。

未来的增强功能

MDB 为开发高级事后分析工具打下了稳定的基础。每个 Solaris 操作系统发行版都包括提供更复杂功能的附加 MDB 模块，用于调试内核和其他软件程序。可以使用 MDB 调试现有的软件程序，以及开发自定义的模块以提高调试自己的 Solaris 驱动程序和应用程序的能力。

第 2 章

调试器概念

本节讨论 MDB 设计的重要方面，以及此体系结构带来的益处。

生成块

目标是指调试器正在检查的程序。 目前 MDB 提供了对以下目标类型的支持：

- 用户进程
- 用户进程核心转储文件
- 没有内核执行控制的实时操作系统（通过 `/dev/kmem` 和 `/dev/ksyms`）
- 具有内核执行控制的实时操作系统（通过 `kmdb(1)`）
- 操作系统崩溃转储
- 在操作系统崩溃转储内记录的用户进程映像
- ELF 目标文件
- 原始数据文件

每个目标都导出标准属性集，包括一个或多个地址空间、一个或多个符号表、一组装入对象和一组线程。图 2–1 概述了 MDB 体系结构，包括两个内置目标和一对样例模块。

在 MDB 术语中，调试器命令又名 **dcmd**（读作 *dee-command*），是调试器中能够访问当前目标的任何属性的例程。 MDB 解析标准输入中的命令，然后执行对应的 dcmd。 每个 dcmd 还可以接受字符串或数值参数的列表，如第 19 页中的 “语法”所示。MDB 包含一组始终可用的内置 dcmd（在第 5 章中介绍）。 通过使用 MDB 附带的编程 API 编写 dcmd，程序员还可以扩展 MDB 本身的功能。

walker 是说明如何遍历或迭代特定程序数据结构的元素的一组例程。 walker 从 dcmd 和从 MDB 本身封装数据结构的实现。 可以交互使用 walker，或者将它们用作元语以生成其他 dcmd 或 walker。 与使用 dcmd 一样，程序员可以通过将其他 walker 作为调试器模块的一部分进行实现来扩展 MDB。

调试器模块又名 **dmod**（读作 *dee-mod*），是包含一组 dcmd 和 walker 的动态装入的库。 在初始化过程中，MDB 尝试装入与目标中存在的装入对象对应的 dmod。 随后可以在运行 MDB 时随时装入或卸载 dmod。 MDB 提供了一组标准 dmod，用于调试 Solaris 内核。

宏文件是包含一组要执行的命令的文本文件。宏文件通常用于自动执行显示简单数据结构的过程。 MDB 提供了完全的向后兼容性以执行为 adb 编写的宏文件。 因此，Solaris 安装附带的一组宏文件可以与任一工具一起使用。

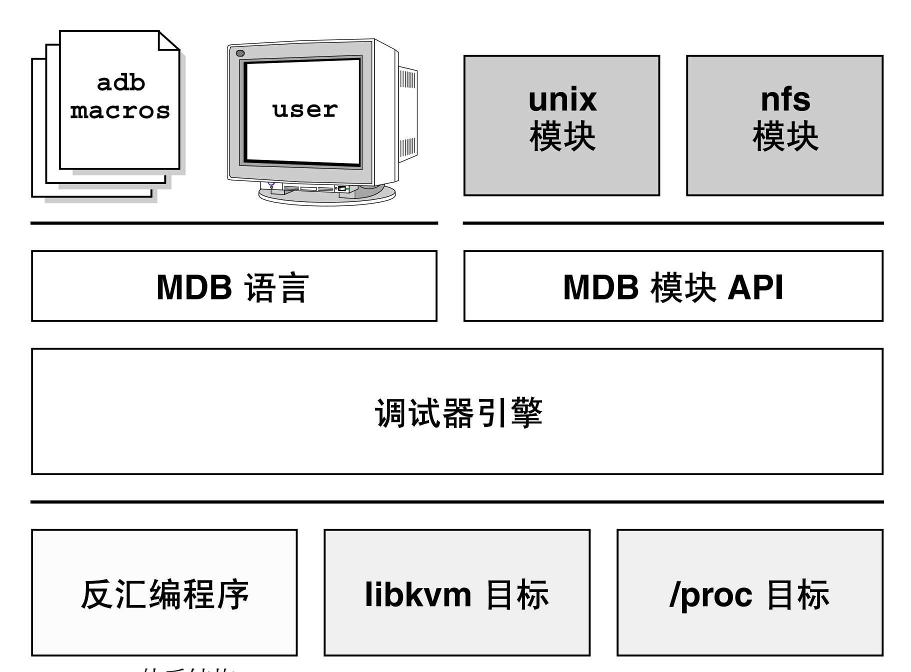


图 2–1 MDB 体系结构

模块化

MDB 的模块化体系结构的优点不仅仅是能够装入包含其他调试器命令的模块。MDB 体系结构各层之间定义了清晰的接口边界（如图 2–1 所示）。宏文件执行以 MDB 或 adb 语言编写的命令。调试器模块中的 Dcmd 和 walker 是使用 MDB 模块 API 编写的，这为允许调试器及其模块独立发展的应用程序二进制接口奠定了基础。

walker 和 dcmd 的 MDB 名称空间还在调试代码（随着目标程序本身的发展，最大限度地实现代码共享并限制必须修改的代码量）之间定义第二组层。例如，Solaris 内核中的主要数据结构之一是表示系统中活动进程的 `proc_t` 结构的列表。`::ps` dcmd 必须迭代此列表才能生成其输出。但是，用于迭代该列表的代码不在 `::ps` dcmd 中，而是封装在 `genunix` 模块的 `proc` walker 中。

MDB 同时提供了 `::ps` 和 `::ptree` dcmd，但是这两者都不知道如何在内核中访问 `proc_t` 结构。相反，它们以程序方式调用 `proc` walker，并相应地格式化返回的结构集。如果用于 `proc_t` 结构的数据结构发生更改，则 MDB 可以提供新的 `proc` walker，而且相关的 dcmd 都不需要更改。 还可以使用 `::walk` dcmd 以交互方式访问 `proc` walker， 以便在调试会话中工作时创建新的命令。

除了便于分层和代码共享外，MDB 模块 API 还提供了具有单一稳定接口的 dcmd 和 walker，以访问基础目标的各种属性。使用同一 API 函数访问用户进程或内核目标中的信息，简化了开发新调试工具的任务。

此外，可以使用自定义 MDB 模块在各种上下文中执行调试任务。例如，您也许希望为正在开发的用户程序开发一个 MDB 模块。开发模块后，当 MDB 检查执行该程序的实时进程、该程序的核心转储、甚至是在执行该程序的系统上所取得的内核崩溃转储时，就可以使用此模块。

模块 API 提供了用于访问以下目标属性的工具：

地址空间　　模块 API 提供了用于从目标的虚拟地址空间读取和写入数据的工具。 还为内核调试模块提供了使用物理地址读取和写入的功能。

符号表　　模块 API 提供了对目标主要可执行文件的静态和动态符号表、其运行时链接编辑器和一组装入对象（用户进程中的共享库或 Solaris 内核中的可装入模块）的访问。

外部数据　　模块 API 提供了用于检索与目标关联的指定外部数据缓冲区集合的工具。 例如，MDB 提供了对与用户进程或用户核心转储文件目标关联的 `proc(4)` 结构的程序访问。

此外，可以使用内置 MDB dcmd 访问有关目标内存映射、装入对象、寄存器值的信息以及控制用户进程目标的执行。

第 3 章

语言语法

本章介绍 MDB 语言语法、运算符以及命令和符号名称解析的规则。

语法

调试器处理来自标准输入的命令。如果标准输入是终端，则 MDB 提供终端编辑功能。MDB 还可以处理来自宏文件和 dcmd 管道的命令，如下所述。语言语法是围绕计算表达式的值（通常为目标中的内存地址）和向该地址应用 dcmd 的概念设计的。当前的地址位置称为 *dot*，"." 用于引用其值。

metacharacter 是以下字符之一：

```
[ ] | ! / \ ? = > $ : ; NEWLINE SPACE TAB
```

blank 是 `TAB` 或 `SPACE`。*word* 是由一个或多个不带引号的元字符分隔的字符序列。一些元字符仅用作某些上下文中的分隔符，如下所述。*identifier* 是字母、数字、下划线、句点或反引号的序列，以字母、下划线或句点开头。标识符用作符号、变量、dcmd 和 walker 的名称。命令由 `NEWLINE` 或分号 (`;`) 分隔。

dcmd 由以下字或元字符之一表示：

```
/ \ ? = > $character :character ::identifier
```

由元字符命名或者前缀为单个 `$` 或 `:` 的 dcmd 是作为内置运算符提供的，实现与传统 `adb(1)` 实用程序命令集的完全兼容。解析 dcmd 后，不再将 `/`、`\`、`?`、`=`、`>`、`$` 和 `:` 字符识别为元字符，直到参数列表结束。

simple-command 是一个后跟零个或多个空格分隔字序列的 dcmd。将字作为被调用 dcmd 的参数进行传递，第 21 页中的 “算术展开”和第 23 页中的 “加引号”中指定的除外。每个 dcmd 都返回一种退出状态，指示它成功、失败或是通过无效参数调用的。

pipeline 是由 `|` 分隔的一个或多个简单命令的序列。与 shell 不同，MDB 管道中的 dcmd 不作为单独的进程执行。解析管道后，按从左到右的顺序调用每个 dcmd。按第 26 页中的

“dcmd 管道”中所述处理和存储每个 dcmd 的输出。左侧 dcmd 完成后，它的输出经处理后用作管道中下一个 dcmd 的输入。如果任何 dcmd 未返回成功退出状态，则异常中止管道。

expression 是对其求值以计算 64 位无符号整数值的字的序列。对这些字求值使用的是第 21 页中的“算术展开”中所述的规则。

命令

command 是以下形式之一：

pipeline [! word ...] [;]
: 简单命令或管道可以选择使用 ! 字符作为后缀，指示调试器应该打开 `pipe(2)`，并将 MDB 管道中最后一个 dcmd 的标准输出发送到通过执行 `$SHELL -c`（后跟由 ! 字符之后的字串联而成的字符串）而创建的外部进程。有关更多详细信息，请参阅第 23 页中的“Shell 转义”。

expression pipeline [! word ...] [;]
: 简单命令或管道可以使用表达式作为前缀。执行管道之前，将点（由“`.`”表示的变量）的值设置为表达式的值。

expression , expression pipeline [! word ...] [;]
: 简单命令或管道可以使用两个表达式作为前缀。计算第一个表达式的值可确定点的新值，计算第二个表达式的值可确定管道中第一个 dcmd 的重复计数。在执行管道中的下一个 dcmd 之前，将对此 dcmd 执行 *count* 次。重复计数仅适用于管道中的第一个 dcmd。

expression pipeline [! word ...] [;]
: 如果省略初始表达式，则不修改点；但是，将根据表达式的值重复管道中的第一个 dcmd。

expression [! word ...] [;]
: 命令只能包含算术表达式。计算表达式的值，将点变量设置为该表达式的值，然后使用点的新值执行前面的 dcmd 和参数。

expression , expression [! word ...] [;]
: 命令只能包含点表达式和重复计数表达式。将点设置为第一个表达式的值后，按第二个表达式的值所指定的次数重复执行前面的 dcmd 和参数。

expression [! word ...] [;]
: 如果省略初始表达式，则不修改点，但是按计数表达式的值所指定的次数重复执行前面的 dcmd 和参数。

! word ... [;]
: 如果命令以 ! 字符开头，则不执行任何 dcmd，而且调试器执行后跟字符串的 `$SHELL -c`，该字符串通过串联 ! 字符后的字而构成的。

注释

字以 `//` 开头时，会忽略该字之后 `NEWLINE` 之前的所有字符。

算术展开

MDB 命令前面是表示起始地址的可选表达式或起始地址和重复计数时，将执行算术展开。也可以执行算术展开以计算 dcmd 的数值参数。算术表达式可以出现在用方括号括起来且前面是美元符号 (`$[ expression ]`) 的参数列表中，并将被替换为表达式的值。

表达式可以包含以下任一特殊字：

integer 指定的整数值。整数值可以使用 `0i` 或 `0I` 作为前缀以指示二进制值，使用 `0o` 或 `0O` 作为前缀以指示八进制值，使用 `0t` 或 `0T` 作为前缀以指示十进制值，以及使用 `0x` 或 `0X` 作为前缀以指示十六进制值（缺省值）。

0[tT][0-9]+.[0-9]+ 指定的十进制浮点值，已转换为其 IEEE 双精度浮点表示形式

'*cccccccc*' 通过将每个字符转换为等于其 ASCII 值的字节而计算的整数值。在字符常量中，最多可以指定八个字符。字符按相反顺序（从右到左）填入整数，从最低有效字节开始。

<*identifier* 由 *identifier* 指定的变量值

identifier 由 *identifier* 指定的符号值

(*expression*) *expression* 的值

. 点值

& 用于执行 dcmd 的最新点值

+ 按当前增量递增的点值

^ 按当前增量递减的点值

增量是一个全局变量，它存储上一个格式设置 dcmd 读取的总字节。有关增量的更多信息，请参阅第 26 页中的 “格式设置 dcmd”的讨论。

一元运算符

一元运算符是右关联的，其优先级高于二元运算符。一元运算符如下：

#expression 逻辑否定

~expression 按位补码

-expression 整型否定

%*expression*	与目标虚拟地址空间中虚拟地址 *expression* 相对应的目标文件位置处指针大小量的值
%/[csil]/*expression*	与目标虚拟地址空间中虚拟地址 *expression* 相对应的目标文件位置处 char、short、int 或 long 大小量的值
%/[1248]/*expression*	与目标虚拟地址空间中虚拟地址 *expression* 相对应的目标文件位置处单字节、双字节、四字节或八字节量的值
**expression*	目标虚拟地址空间中虚拟地址 *expression* 处指针大小量的值
*/[csil]/*expression*	目标虚拟地址空间中虚拟地址 *expression* 处 char、short、int 或 long 大小量的值
*/[1248]/*expression*	目标虚拟地址空间中虚拟地址 *expression* 处单字节、双字节、四字节或八字节量的值

二元运算符

二元运算符是左关联的，其优先级低于一元运算符。按最高到最低的优先顺序排列，二元运算符如下：

`*`	整数相乘
`%`	整数相除
`#`	将左侧值向上舍入为最接近的右侧值倍数
`+`	整数相加
`-`	整数相减
`<<`	按位向左移位
`>>`	按位向右移位
`==`	逻辑相等
`!=`	逻辑不等
`&`	按位和
`^`	按位异或
`|`	按位或

加引号

前面介绍的每个元字符（请参见第 19 页中的 “语法”）都会终止字，除非用引号引起来。通过将字符放在一对单引号 (') 或双引号 (") 之中，可以将其引起来，以强制 MDB 将每个字符解释为其本身，而没有任何特殊意义。单引号不能出现在单引号内。在双引号内，MDB 可识别 C 编程语言字符转义序列。

Shell 转义

可以使用 ! 字符创建 MDB 命令和用户的 shell 之间的管道。Shell 转义仅在使用 mdb（而不是 kmdb）时可用。如果设置了 $SHELL 环境变量，则 MDB 将为 shell 转义对此程序执行 fork 和 exec；否则使用 /bin/sh。shell 是使用后跟字符串的 -c 选项调用的，该字符串通过串联 ! 字符后的字而构成。

! 字符优先于所有其他元字符，但分号 (;) 和 NEWLINE 除外。检测到 shell 转义后，下一个分号或 NEWLINE 之前的其余字符将“按原样”传递到 shell。shell 命令的输出不能传输到 MDB dcmd。shell 转义执行的命令将其输出直接发送到终端而不是 MDB。

变量

variable 是变量名称、对应的整数值和一组属性。变量名称是字母、数字、下划线或句点的序列。可以使用 > dcmd 或 ::typeset dcmd 为变量赋值，而且可以使用 ::typeset dcmd 处理其属性。每个变量的值都表示为 64 位无符号整数。变量可以具有一个或多个以下属性：只读（用户不能进行修改）、持久性（用户不能取消设置）和带有标记（用户定义的指示符）。

以下变量被定义为持久变量：

0	使用 /、\、? 或 = dcmd 列显的最新值
9	用于 $< dcmd 的最新计数
b	数据节基数的虚拟地址
cpuid	与当前执行 kmdb 的 CPU 相对应的 CPU 标识符。
d	数据节的大小（以字节为单位）
e	入口点的虚拟地址
hits	与软件事件说明符匹配的次数。请参见第 48 页中的 “事件回调”。
m	目标的主目标文件的初始字节（魔数）；如果尚未读取目标文件，则为零
t	文本节的大小（以字节为单位）

thread　当前代表线程的线程标识符。标识符的值取决于当前目标所用的线程模型。请参见第 48 页中的 “线程支持” 。

此外，MDB 内核和进程目标将代表线程的寄存器集的当前值导出为命名变量。这些变量的名称取决于目标的平台和指令集体系结构。

符号名称解析

如第 19 页中的 “语法”中所述，表达式上下文中存在的符号标识符的计算结果是此符号的值。该值通常表示与目标虚拟地址空间中符号关联的存储区的虚拟地址。目标可以支持多个符号表，包括但不限于：

- 主可执行文件符号表
- 主动态符号表
- 运行时链接编辑器符号表
- 许多装入对象（如用户进程中的共享库或 Solaris 内核中的内核模块）中每个装入对象的标准符号表和动态符号表

目标通常先搜索主可执行文件的符号表，然后搜索一个或多个其他符号表。请注意，ELF 符号表仅包含外部、全局和静态符号的项；自动符号不出现在 MDB 处理的符号表中。

此外，MDB 提供了专用的用户定义符号表，在搜索任何目标符号表之前都会先搜索该表。专用符号表最初为空，可以使用 `::nmadd` 和 `::nmdel` dcmd 处理它。

可以使用 `::nm -P` 选项显示专用符号表的内容。通过专用符号表，用户可以为程序函数或者原始程序中缺少的或已删除的数据创建符号定义。然后，每当 MDB 将符号名称转换为地址，或将地址转换为最接近的符号时，都可以使用这些定义。

由于目标包含多个符号表，而且每个符号表可以包括多个目标文件中的符号，因此可能存在同名的不同符号。MDB 将反引号 “‘“ 字符用作符号名称作用域运算符，以允许程序员在这种情况下获取所需符号的值。

可以指定用于将符号名称解析为以下任一内容的作用域：*object*‘*name*、*file*‘*name* 或 *object*‘*file*‘*name*。对象标识符引用装入对象的名称。文件标识符引用源文件的基本名称，该文件在指定对象的符号表中具有 `STT_FILE` 类型的符号。如何解释对象标识符取决于目标类型。

MDB 内核目标要求 *object* 指定已装入内核模块的基本名称。例如，符号名称：

```
specfs‘_init
```

的计算结果是 `specfs` 内核模块中 `_init` 符号的值。

`mdb` 进程目标要求 *object* 指定可执行文件或已装入共享库的名称。它可以采用以下任一形式：

- 完全匹配（即，完整路径名）：/usr/lib/libc.so.1
- 基本名称完全匹配：libc.so.1
- 与基本名称开头匹配（即取”.”后缀之前的部分）：libc.so 或 libc
- 作为可执行文件的别名接受的文字字符串 a.out

进程目标还可接受上述四种形式前面加上可选的链接映射 id (lmid)。 lmid 前缀由初始 LM 后跟十六进制链接映射 id 以及附加反引号指定。 例如，符号名称：

```
LM0‘libc.so.1‘_init
```

的计算结果将是在链接映射 0 (LM_ID_BASE) 上装入的 libc.so.1 库中 _init 符号的值。 如果在多个链接映射上装入同一库，则解决符号命名冲突可能需要链接映射说明符。 有关链接映射的更多信息，请参阅《链接程序和库指南》和 dlopen(3C) 手册页。根据 showlmid 选项的设置列显符号时将显示链接映射标识符，如第 127 页中的 “命令行选项摘要”中所述。

如果符号和十六进制整数值之间存在命名冲突，则 MDB 尝试首先将不明确的标记计算为符号，然后将它计算为整数值。例如，标记 f 可以指以十六进制（缺省基数）指定的十进制整数值 15，也可以指目标符号表中名为 f 的全局变量。如果存在名称不明确的符号，则可以使用显式 0x 或 0X 前缀指定整数值。

dcmd 和 Walker 名称解析

如上所述，每个 MDB dmod 都提供了一组 dcmd 和 walker。dcmd 和 walker 是在两个不同的全局名称空间中跟踪的。MDB 还跟踪与每个 dmod 关联的 dcmd 和 walker 名称空间。在给定 dmod 内不允许存在名称完全相同的 dcmd 或 walker：具有此类型命名冲突的 dmod 将无法装入。

在全局名称空间中，允许在不同 dmod 的 dcmd 或 walker 之间存在名称冲突。 如果存在冲突，则优先装入全局名称空间中具有该特定名称的第一个 dcmd 或 walker。替换定义按装入顺序保存在列表中。

可以将反引号字符 “‘“ 用作 dcmd 或 walker 名称中的作用域运算符以选择替换定义。例如，如果 dmod m1 和 m2 都提供了 dcmd d，而且 m1 是在 m2 之前装入的，则：

::d	执行 m1 的 d 定义
::m1‘d	执行 m1 的 d 定义
::m2‘d	执行 m2 的 d 定义

如果当前卸载了模块 m1，则全局定义列表上的下一个 dcmd (m2‘d) 将被提升为全局可见。可以使用 ::which dcmd 确定 dcmd 或 walker 的当前定义，如下所述。可以使用 ::which -v 选项显示全局定义列表。

dcmd 管道

使用 | 运算符可以将 dcmd 编入管道中。管道的用途是将值列表（通常为虚拟地址）从一个 dcmd 或 walker 传递到另一个 dcmd 或 walker。可以使用管道阶段将一种数据结构类型的指针映射到指向对应数据结构的指针，对地址列表排序，或者选择具有某些属性的结构的地址。

MDB 按从左向右的顺序执行管道中的每个 dcmd。最左侧的 dcmd 是使用点的当前值执行的，或者是使用命令开头的显式表达式指定的值执行的。遇到 | 运算符时，MDB 将在其左侧的 dcmd 输出和 MDB 解析器之间创建一个管道（共享缓冲区）以及一个空白值列表。

在 dcmd 执行时，其标准输出放置在管道中，然后由解析器使用和计算，好像 MDB 从标准输入读取此数据。每行都必须包含以 `NEWLINE` 或分号 (`;`) 结尾的算术表达式。将表达式的值附加到与管道关联的值列表。如果检测到语法错误，则中止管道。

在 | 运算符左侧的 dcmd 完成后，使用与管道关联的值列表调用 | 运算符右侧的 dcmd。对于列表中的每个值，将点设置为此值并执行右侧的 dcmd。只有管道中最右侧的 dcmd 才将其输出列显到标准输出。如果管道中的任何 dcmd 产生标准错误输出，则这些消息将直接列显为标准错误，而不作为管道的一部分进行处理。

格式设置 dcmd

`/`、`\`、`?` 和 `=` 元字符用于表示特殊的输出格式设置 dcmd。其中的每个 dcmd 都接受由一个或多个格式字符、重复计数或带引号字符串组成的参数列表。格式字符是下表所示的 ASCII 字符之一。

格式字符用于从目标读取数据和设置数据的格式。重复计数是格式字符前面的正整数，始终用基数 10（十进制）解释它。也可以将重复计数指定为括在方括号中、前面是美元符号的表达式 (`$[ ]`)。必须用双引号 (`" "`) 将字符串参数引起来。格式参数之间不需要留空格。

格式设置 dcmd 包括：

`/`
显示目标的虚拟地址空间（从点指定的虚拟地址开始）中的数据。

`\`
显示目标的物理地址空间（从点指定的物理地址开始）中的数据。

`?`
显示目标的主目标文件中从与点指定的虚拟地址相对应的目标文件位置开始的数据。

`=`
按每种指定的数据格式显示点本身的值。因此 `=` dcmd 对于在基数之间转换和执行运算是很有用的。

除了点外，MDB 还跟踪名为 *increment* 的另一全局值。增量表示点和上一格式设置 dcmd 读取的所有数据后面的地址之间的距离。

例如，如果格式设置 dcmd 是在点等于地址 A 的情况下执行的，且显示一个 4 字节整数，则在此 dcmd 完成后，点仍为 A，但是增量设置为 4。第 21 页中的 “算术展开”中所述的 + 字符的计算结果现在是值 A + 4，并可以用于将点重置为后续 dcmd 的下一个数据对象的地址。

大多数的格式字符按与表中所示的数据格式大小相对应的字节数增加增量的值。使用 ::formats dcmd 可以从 MDB 内显示格式字符表。

格式字符包括：

+
: 按计数递增点（可变大小）

-
: 按计数递减点（可变大小）

B
: 十六进制 int（1 字节）

C
: 使用 C 字符表示法的字符（1 字节）

D
: 十进制带符号 int（4 字节）

E
: 十进制无符号 long long（8 字节）

F
: 双精度（8 字节）

G
: 八进制无符号 long long（8 字节）

H
: 交换字节和 short（4 字节）

I
: 地址和反汇编的指令（可变大小）

J
: 十六进制 long long（8 字节）

K
: 十六进制 uintptr_t（4 或 8 字节）

N
: 新行

O
: 八进制无符号 int（4 字节）

P
: 符号（4 或 8 字节）

Q
: 八进制带符号 int（4 字节）

R
: 二进制 int（8 字节）

S
: 使用 C 字符串表示法的字符串（可变大小）

T
: 水平制表符

U
: 十进制无符号 int（4 字节）

V
: 十进制无符号 int（1 字节）

W
: 缺省基数无符号 int（4 字节）

X
: 十六进制 int（4 字节）

Y
: 已解码的 time32_t（4 字节）

Z
: 十六进制 long long（8 字节）

^
: 按增量 * 计数递减点（可变大小）

a
: 点作为符号+偏移

b
: 八进制无符号 int（1 字节）

c
: 字符（1 字节）

d
: 十进制带符号 short（2 字节）

e
: 十进制带符号 long long（8 字节）

f
: 浮点（4 字节）

g
: 八进制带符号 long long（8 字节）

h
交换字节（2 字节）

i
反汇编的指令（可变大小）

n
新行

o
八进制无符号 short（2 字节）

p
符号（4 或 8 字节）

q
八进制带符号 short（2 字节）

r
空格

s
原始字符串（可变长度）

t
水平制表符

u
十进制无符号 short（2 字节）

v
十进制带符号 int（1 字节）

w
缺省基数无符号 short（2 字节）

x
十六进制 short（2 字节）

y
已解码的 time64_t（8 字节）

/、\ 和 ? 格式设置 dcmd 还可以用于写入目标的虚拟地址空间、物理地址空间或目标文件，方法是将以下修饰符之一指定为第一个格式字符，然后指定作为即时值或括在方括号中且前面是美元符号的表达式 ($[ ]) 的字的列表。

写入修饰符包括：

v
将每个表达式值的最低字节写入从点指定的位置开始的目标

w
将每个表达式值的最低 2 字节写入从点指定的位置开始的目标

W
: 将每个表达式值的最低 4 字节写入从点指定的位置开始的目标

Z
: 将每个表达式值的完整 8 字节写入从点指定的位置开始的目标

/、\ 和 ? 格式设置 dcmd 还可以分别用来在搜索目标的虚拟地址空间、物理地址空间和目标文件中搜索特定的整数值，方法是将以下修饰符之一指定为第一个格式字符，然后指定值和可选掩码。值和掩码均指定为即时值或括在方括号中且前面是美元符号的表达式。

如果仅指定了值，则 MDB 读取相应大小的整数，并停止在包含匹配值的位置。如果指定了值 V 和掩码 M，则 MDB 读取相应大小的整数，并停止在包含值 X（其中 (X & M) == V）的位置。dcmd 完成时，点将更新为包含匹配项的地址。如果找不到匹配项，则点停留在读取的最后一个地址处。

搜索修饰符包括：

l
: 搜索指定的 2 字节的值

L
: 搜索指定的 4 字节的值

M
: 搜索指定的 8 字节的值

对于用户目标和内核目标，地址空间通常都由一组不连续的段组成。从没有对应段的地址中读取是非法的。如果搜索到达了段边界但未找到匹配项，则在超过段边界结尾的读取失败时它将异常中止。

◆ ◆ ◆ 第 4 章

交互

本章介绍 MDB 交互式命令行编辑和历史记录功能、输出页面调度程序以及调试器信号处理。

命令重新输入

从终端设备输入的最后 HISTSIZE（缺省值为 128）个命令的文本 保存在内存中。接下来将介绍的内嵌编辑工具提供了用于从历史记录列表中搜索和提取元素的键映射。

内嵌编辑

如果标准输入是终端设备，则 MDB 将提供一些简单的 emacs 样式工具，以用于 编辑命令行。在编辑模式下使用 search、previous 和 next 命令可访问历史记录列表。搜索时仅匹配字符串而不匹配模式。在以下列表中，控制字符的符号为插入记号 (^) 后跟一个大写字符。转义序列的符号为 M- 后跟一个字符。例如，M-f（读作 *meta- eff*）是通过按下 <ESC> 再按下 'f' 输入的，或者在支持 Meta 键的键盘上按下 Meta 再按下 'f' 输入。命令行是使用回车符或换行符提交和执行的。编辑命令包括：

^F 将光标前（右）移一个字符。

M-f 将光标前移一个字。

^B 将光标后（左）移一个字符。

M-b 将光标后移一个字。

^A 将光标移至行首。

^E 将光标移至行尾。

^D 如果当前行不为空，则删除当前字符。如果当前行为空，则 ^D 表示 EOF 并且调试器将退出。

M-^H （Meta-backspace 组合键）删除前一个字。

^K　　删除从光标到行尾的内容。

^L　　重新列显当前行。

^T　　将当前字符与下一个字符换位。

^N　　从历史记录中提取下一个命令。每次输入 ^N 后，检索时间靠前的下一个命令。

^P　　从历史记录中提取上一个命令。每次输入 ^P 后，检索时间靠后的下一个命令。

^R[string]　　在历史记录中向后搜索包含字符串的上一个命令行。该字符串应该以回车符或换行符结尾。如果省略字符串，则将检索包含最新字符串的上一个历史记录元素。

编辑模式还会将以下用户定义的序列解释为编辑命令。可以使用 `stty(1)` 命令读取或修改用户定义的序列。

erase　　用户定义的删除字符（通常为 ^H 或 ^?），用于删除上一个字符。

intr　　用户定义的中断字符（通常为 ^C）， 用于异常中止当前命令并列显新提示。

kill　　用户定义的中止字符（通常为 ^U）， 用于中止整个当前命令行。

quit　　用户定义的退出字符（通常为 ^\）， 退出调试器。

suspend　　用户定义的暂停字符（通常为 ^Z）， 用于暂停调试器。

werase　　用户定义的字删除字符（通常为 ^W），用于删除前一个字。

在支持 带方向键的扩展小键盘的键盘上，`mdb` 会将这些击键解释为编辑命令：

向上方向键　　从历史记录中提取上一个命令（与 ^P 相同）。

向下方向键　　从历史记录中提取下一个命令（与 ^N 相同）。

向左方向键　　将光标后移一个字符（与 ^B 相同）。

向右方向键　　将光标前移一个字符（与 ^F 相同）。

快捷键

MDB 提供了一组快捷键，可在 MDB 提示符后面将下表列出的击键作为第一个字符键入时，将单个击键绑定到常用的 MDB 命令。快捷键包括：

`[`
　执行命令 `::step over`

`]`
　执行命令 `::step`

输出页面调度程序

mdb 提供了内置的 输出页面调度程序。如果调试器的标准输出是终端设备，则会启用输出页面调度程序。每次执行命令时，mdb 都会在生成一满屏输出后暂停，并会显示页面调度程序提示：

```
>> More [<space>, <cr>, q, n, c, a] ?
```

页面调度程序可识别以下键序：

空格键	显示下一满屏输出。
a、A	异常中止当前顶级命令并返回到提示符。
c、C	连续显示输出而不会在每一满屏时暂停，直到完成当前顶级命令为止。
n、N、NEWLINE、RETURN	显示下一行输出。
q、Q、^C、^\	仅退出（异常中止）当前的 dcmd。

信号处理

调试器会忽略 PIPE 和 QUIT 信号。INT 信号会异常中止当前执行的命令。调试器可拦截 ILL、TRAP、EMT、FPE、BUS 和 SEGV 信号，并提供相应的特殊处理。如果其中任一信号是以异步方式生成的（即使用 kill(2) 从其他进程传送），则 mdb 会将该信号恢复为其缺省部署和转储核心。但是，如果其中任一信号是由调试器进程本身同步生成的，从外部装入 dmod 中的 dcmd 当前正在执行并且标准输入是终端，则 mdb 将提供一个选项菜单，允许用户强制执行核心转储、退出而不生成核心转储、为调试器附加而停止或尝试恢复等操作。 恢复选项将异常中止所有活动的命令，并卸载出现故障时其 dcmd 处于活动状态的 dmod。然后，用户可以将其重新装入。恢复选项可针对错误的 dcmd 提供有限的保护。有关与恢复选项关联的风险的信息，请参阅第 133 页中的 “警告”中的“使用错误恢复机制”部分。

◆ ◆ ◆ 第 5 章

内置命令

MDB 提供了一组始终定义的内置 dcmd。其中一些 dcmd 仅适用于某些目标：如果 dcmd 不适用于当前目标，则它将失败并列显一条消息，指示“当前目标不支持该命令”。

在许多情况下，MDB 为传统的 `adb(1)` dcmd 名称提供了助记等效项（`::identifier`）。例如，提供了 `::quit` 作为 `$q` 的等效项。对 `adb(1)` 有经验或欣赏简洁或深奥的程序员可能会首选 `$` 或 `:` 形式的内置命令。不熟悉 MDB 的程序员可能会首选更详细的 `::` 形式。内置命令按字母顺序显示。如果 `$` 或 `:` 形式具有 `::identifier` 等效项，则以 `::identifier` 形式显示它。

内置 dcmd

> *variable-name*

> /*modifier*/ *variable-name*

将点的值赋给指定的命名变量。一些变量是只读的，无法进行修改。如果 > 后跟置于 `//` 中的修饰符，则在赋值过程中修改该值。修饰符如下：

`c`

无符号 char 值（1 字节）

`s`

无符号 short 值（2 字节）

`i`

无符号 int 值（4 字节）

`l`

无符号 long 值（在 32 位环境中为 4 字节，在 64 位环境中为 8 字节）

请注意，这些运算符不执行强制类型转换；相反，它们提取指定的低位字节数（在小端字节序体系结构中）或高位字节数（在大端字节序体系结构中）。提供这些修饰符是为了向下兼容；应该改用 MDB */*modifier*/ 和 %/*modifier*/ 语法。

`$<` *macro-name*
从指定的宏文件读取和执行命令。可以以绝对路径或相对路径指定文件名。如果文件名是简单名称（即，如果它不包含'/'），则 MDB 会在宏文件包含路径中搜索它。如果当前正在处理其他宏文件，则关闭此文件并将其替换为新文件。

`$<<` *macro-name*
从指定的宏文件读取和执行命令（与 `$<` 一样），但不关闭当前打开的宏文件。

`$?`
列显目标的进程 ID 和当前信号（如果该目标是用户进程或核心转储文件），然后列显代表线程的典型寄存器集。

[*address*] `$C` [*count*]
列显 C 栈反向跟踪，包括栈帧指针信息。如果在 dcmd 的前面明确指定了 *address*，则显示从此虚拟内存地址开始的反向跟踪。否则，显示代表线程的栈。如果提供了可选计数值作为参数，则在输出中为每个栈帧显示的参数不超过 *count* 个。

仅适用于 64-bit SPARC – 请求栈跟踪时，应该将偏离的帧指针值（即，虚拟地址减去 `0x7ff`）用作地址。

[*base*] `$d`
获取或设置缺省的输出基数。如果在 dcmd 的前面明确指定了表达式，则将缺省输出基数设置为给定的 *base*；否则，以基数 10（十进制）列显当前基数。缺省基数是基数 16（十六进制）。

`$e`
列显关于类型对象或函数的所有已知外部（全局）符号、符号的值以及在目标虚拟地址空间中此位置上存储的前 4 个（32 位 `mdb`）或前 8 个（64 位 `mdb`）字节的列表。`::nm` dcmd 提供了用于显示符号表的更灵活选项。

`$P` *prompt-string*
将提示符设置为指定的 *prompt-string*。缺省提示符是'`>`'。也可以使用 `::set -P` 或 `-P` 命令行选项设置提示符。

`$M`
仅在 `kmdb` 中，列出由 `kmdb` 高速缓存，以便与 `$<` dcmd 一起使用的宏文件。

distance `$s`
为地址到符号名称的转换获取或设置符号匹配 *distance*。符号匹配距离模式将与附录 A中的 `-s` 命令行选项一起讨论。也可以使用 `::set -s` 选项修改符号匹配距离。如果未指定距离，则显示当前设置。

`$v`
列显具有非零值的命名变量的列表。`::vars` dcmd 提供了用于列出变量的其他选项。

width `$w`
将输出页 *width* 设置为指定值。通常，此命令不是必需的，因为 MDB 会向终端查询其宽度并处理调整大小事件。

`$W`
　重新打开目标以写入，就像已在命令行上使用 `-w` 选项执行了 MDB。也可以使用 `::set -w` 选项启用写入模式。

`::array` *type count*
　列显数组的每个元素的地址。应该将数组元素的类型指定为第一个参数 *type*，而且应该将要计算的元素数指定为第二个参数 *count*。可以将 `::array` 的输出传输到 `::print` dcmd，以列显数组数据结构的元素。

注 – 此 dcmd 只能与包含压缩的符号调试信息的对象（设计用于 `mdb`）一起使用。此信息当前仅可用于某些 Solaris 内核模块。必须安装 SUNWzlib 解压缩软件，才能处理符号调试信息。

[*pid*] `::attach` [*core* | *pid*]
[*pid*] `:A` [*core* | *pid*]
　如果用户进程目标处于活动状态，则附加到指定的进程 ID 或 *core* 文件并调试它。应该将核心转储文件路径名指定为字符串参数。可以将进程 ID 指定为字符串参数，也可以指定为 dcmd 前面的表达式的值。回想一下，缺省基数是十六进制的，因此，使用 `pgrep(1)` 或 `ps(1)` 获取的十进制 PID 在被指定为表达式时其前面应该有 "`0t`"。

`::branches` [`-v`]
　显示当前 CPU 采用的最新分支。当前，仅在启用相应处理器特定功能的 x86 系统上使用 `kmdb` 时，此 dcmd 才可用。可以显示的分支数和分支类型由处理器体系结构确定。如果存在 `-v` 选项，则显示每个分支前面的指令。

`::cat` *filename ...*
　串联并显示文件。可以将每个文件名指定为相对路径名或绝对路径名。文件内容将列显到标准输出，但是不会通过输出页面调度程序。此 dcmd 旨在与 `|` 运算符一起使用；程序员可以使用存储在外部文件中的地址列表启动管道。

address `::context`
address `$p`
　将上下文切换到指定的进程。上下文切换操作仅在使用内核目标时有效。进程上下文是在内核虚拟地址空间中使用其 `proc` 结构的 *address* 指定的。特殊上下文地址 "`0`" 用于表示内核本身的上下文。仅当转储包含指定用户进程的物理内存页面（与仅包含内核页面相对）时，MDB 才能在检查崩溃转储时执行上下文切换。使用 `dumpadm(1M)`，可以将内核崩溃转储工具配置为转储所有页面或当前用户进程的页面。`::status` dcmd 可以用于显示当前崩溃转储的内容。

　当用户请求从内核目标进行上下文切换时，MDB 将构造一个表示指定用户进程的新目标。发生切换后，新目标将在全局级别上插入其 dcmd：因此，现在 `/` dcmd 可以设置用户进程虚拟地址空间中数据的格式并显示该数据，`::mappings` dcmd 可以显示用户进程的地址空间中的映射，依此类推。通过执行 `0::context` 可以恢复内核目标。

`::cpuregs` [`-c` *cpuid*]
　显示当前 CPU 或指定 *cpuid* 的当前典型寄存器集。此命令仅在使用 `kmdb` 时可用。

`::cpustack` [`-c` *cpuid*]
显示在当前 CPU 或指定 *cpuid* 上执行的线程的 C 栈反向跟踪。此命令仅在使用 `kmdb` 时可用。

`::dcmds`
列出可用的 dcmd 并列显其中每个 dcmd 的简短说明。

[*address*] `::dis` [`-afw`] [`-n` *count*] [*address*]
从最后一个参数或点的当前值指定的 *address* 处或其附近开始进行反汇编。如果地址与已知函数的开头匹配，则反汇编整个函数；否则，列显指定地址前后的指令的"窗口"，以便提供上下文。缺省情况下，从目标的虚拟地址空间读取指令；如果存在 `-f` 选项，则改为从目标的目标文件读取指令。如果调试器当前未附加到实时进程、核心转储文件或崩溃转储，则缺省情况下启用 `-f` 选项。`-w` 选项可以用于强制窗口模式，即使地址是已知函数的开头，也是如此。窗口大小缺省为十条指令；可以使用 `-n` 选项显式指定指令数。如果存在 `-a` 选项，则以数值而不是符号形式列显地址。

`::disasms`
列出可用的反汇编程序模式。初始化目标时，MDB 尝试选择相应的反汇编程序模式。用户可以使用 `::dismode` dcmd 将模式更改为列出的任一模式。

`::dismode` [*mode*]
`$V` [*mode*]
获取或设置反汇编程序模式。如果未指定参数，则列显当前的反汇编程序模式。如果指定了 *mode* 参数，则将反汇编程序切换到指定模式。可以使用 `::disasms` dcmd 显示可用反汇编程序的列表。

`::dmods` [`-l`] [*module-name*]
列出已装入的调试器模块。如果指定了 `-l` 选项，则将在每个 dmod 的名称下列显与其关联的 dcmd 和 walker 的列表。通过将其名称指定为其他参数，可以将输出限制为特定的 dmod。

[*address*] `::dump [- eqrstu] [-f|-p] [-g` *bytes*`] [-w` *paragraphs*`]`
列显 16 字节对齐的虚拟内存区域（包含点所指定的地址）的十六进制和 ASCII 内存转储。如果为 `::dump` 指定了重复计数，则将此解释为要转储的字节数而不是迭代次数。`::dump` dcmd 还可识别以下选项：

`-e`	按字节存储顺序调整。`-e` 选项采用 4 字节字；`-g` 选项可以用于更改缺省的字大小。
`-f`	从对应于给定虚拟地址的目标文件位置读取数据，而不是从目标的虚拟地址空间读取数据。如果调试器当前未附加到实时进程、核心转储文件或崩溃转储，则缺省情况下启用 `-f` 选项。
`-g` *group*	以字节组显示字节。缺省的 *group* 大小为 4 字节。*group* 大小必须是 2 的幂，且可以整除行宽。
`-p`	将 *address* 解释为目标地址空间中的物理地址位置，而不是虚拟地址。
`-q`	不列显数据的 ASCII 解码。
`-r`	相对于起始地址而不是每行的显式地址的数字行。此选项隐含 `-u` 选项。

`-s` 取消重复行。

`-t` 仅读取并显示指定地址的内容，而不是读取和列显整行。

`-u` 取消对齐输出，而不是在段落边界上对齐输出。

`-w` *paragraphs* 显示段落，每行为 16 字节段落。缺省的段落数为 1。 `-w` 可接受的最大值为 16。

`::echo` [*string* | *value* ...]
将由空格分隔并以 `NEWLINE` 结尾的参数列显到标准输出。将计算括在 `$[ ]` 中的表达式，得出一个值，并以缺省基数列显它。

`::eval` *command*
评估并执行指定的用作命令的字符串。如果命令包含元字符或空格，则应该用双引号或单引号将该命令引起来。

`::files` [*object*]
`$f`
列显已知源文件的列表（各种目标符号表中存在的类型为 `STT_FILE` 的符号）。如果指定了 *object* 名称，则将输出限制为存在于对应目标文件中的文件符号。

`[`*address*`] ::findsym [-g] [` *address* `|` *symbol* `...]`
在指令文本中搜索引用指定的符号或地址的指令。搜索列表应该包含一个或多个指定为 dcmd 前面的地址的地址或符号名称，或者包含 dcmd 后面的一个或多个符号名称或表达式。如果指定了 `-g` 选项，则将搜索限制为目标符号表中作为全局可见函数一部分的指令文本。

注 – **仅限于** *SPARC*。仅当调试使用 SPARC 指令集体系结构的目标时，`::findsym` dcmd 才可用。

`::formats`
列出与 `/`、`\`、`?` 和 `=` 等格式设置 dcmd 一起使用的可用的输出格式字符。这些格式及其用法在第 26 页中的 “格式设置 dcmd”中说明。

`[ thread ] ::fpregs [-dqs]`
`[ thread ] $x, $X, $y, $Y`
列显代表线程的浮点寄存器集。如果指定了一个线程，则显示该线程的浮点寄存器。线程表达式应该为第 48 页中的 “线程支持”下说明的线程标识符之一。

注 – 仅限于 SPARC。 `-d`、`-q` 和 `-s` 选项可以用于将浮点寄存器显示为双精度 (`-d`)、四精度 (`-q`) 或单精度 (`-s`) 浮点值的集合。

`::grep` *command*
评估指定的命令字符串，如果点的新值不为零，则列显点的旧值。如果 *command* 包含空格或元字符，则必须使用引号将命令字符串引起来。可以在管道中使用 `::grep` dcmd 过滤地址列表。

`::help` [*dcmd-name*]
如果不带参数，则 `::help` dcmd 列显 MDB 中可用的帮助工具的简短概述。如果指定了 *dcmd-name*，则 MDB 列显该 dcmd 的用法摘要。

[*address* [, *len*]] `::in` [`-L` *len*]
从 *address* 指定的 I/O 端口读取和显示 *len* 字节。如果存在 `-L` 选项的值，则它优先于在左侧指定的重复计数。*len* 必须为 1、2 或 4 字节，而且端口地址必须按照长度对齐。仅当在 x86 系统上使用 `kmdb` 时，此命令才可用。

[*address*] `::list` *type member* [*variable-name*]
遍历链接列表数据结构的元素，并列显列表中每个元素的地址。可以使用可选的 *address* 指定列表中第一个元素的地址；否则，假定列表从点的当前值开始。*type* 参数必须指定一种 C 结构或联合类型，它用于说明列表元素的类型，以便 MDB 可以读入相应大小的对象。*member* 参数用于指定包含指向下一个列表元素的指针的 *type* 的成员。`::list` dcmd 将继续迭代，直到遇到 NULL 指针、再次到达第一个元素（循环列表）或者读取元素时出错。如果指定了可选的 *variable-name*，则 MDB 调用管道的下一个阶段时，会将在遍历的每一步返回的值赋给指定的变量。

注 – 此 dcmd 只能与包含压缩的符号调试信息的对象（设计用于 `mdb`）一起使用。此信息当前仅可用于某些 Solaris 内核模块。必须安装 SUNWzlib 解压缩软件，才能处理符号调试信息。

`::load` [`-s`] *module-name*
装入指定的 dmod。可以将模块名称指定为绝对路径或相对路径。如果 *module-name* 是简单名称（即，不包含 '/'），则 MDB 将在模块库路径中搜索它。无法装入名称冲突的模块；此时，必须先卸载已有的模块才可继续装入。如果存在 `-s` 选项，则在找不到或无法装入模块时，MDB 将保持无提示状态，不发出任何错误消息。

`::log` [`-d` | [`-e`] *filename*]
`$>` [*filename*]
启用或禁用输出日志。MDB 提供了一种交互式日志记录工具，使用该工具可以在仍与用户交互的同时，将输入命令和标准输出记录到文件中。`-e` 选项允许记录到指定的文件中；如果未指定文件名，则重新允许记录到以前的日志文件中。`-d` 选项禁用日志记录。如果使用了 `$>` dcmd，则在指定文件名参数时启用日志记录；否则，禁用日志记录。如果指定的日志文件已存在，则 MDB 会将任何新日志输出附加到该文件。

`::map` *command*
将指定的 *command* 用作字符串参数，将点的值映射到对应值，然后列显点的新值。如果命令包含空格或元字符，则必须使用引号将此命令引起来。可以在管道中使用 `::map` dcmd 将地址列表转换为新的地址列表。

[*address*] `::mappings` [*name*]
[*address*] `$m` [*name*]
列显目标虚拟地址空间中每个映射的列表，包括每个映射的地址、大小和说明。如果 dcmd 前面有 *address*，则 MDB 仅显示包含给定地址的映射。如果指定了字符串 *name* 参数，则 MDB 仅显示与说明匹配的映射。

[*address*] ::nm [-DPdghnopuvx] [- t *types*] [- f *format*] [*object*]

列显与当前目标关联的符号表。如果指定了 dcmd 前面的可选 *address*，则仅显示对应于该地址的符号的符号表项。如果指定了 *object* 名称，则仅显示此装入对象的符号表。::nm dcmd 还可识别以下选项：

-D	列显 .dynsym（动态符号表）而不是 .symtab。
-P	列显专用符号表而不是 .symtab。
-d	以十进制列显值和大小字段。
-f format [,format...]	仅列显指定的符号信息。有效的格式参数字符串如下：

ndx	符号表索引
val	符号表
size	大小（以字节为单位）
type	符号类型
bind	绑定
oth	其他
shndx	节索引
name	符号名称
ctype	符号的 C 类型（如果已知）
obj	定义符号的对象

-g	仅列显全局符号。
-h	抑制标题行。
-n	按名称对符号排序。
-o	以八进制列显值和大小字段。
-p	将符号列显为一系列 ::nmadd 命令。此选项可以与 -P 一起使用，以生成宏文件（随后可以使用 $< 将其读入调试器）。
-t type [,type...]	仅列显指定类型的符号。有效的类型参数字符串如下：

noty	STT_NOTYPE
objt	STT_OBJECT
func	STT_FUNC
sect	STT_SECTION
file	STT_FILE
comm	STT_COMMON

	tls	STT_TLS
	regi	STT_SPARC_REGISTER
`-u`	仅列显未定义的符号。	
`-v`	按值对符号排序。	
`-x`	以十六进制列显值和大小字段。	

value `::nmadd` [`-fo`] [`-e` *end*] [`-s` *size*] *name*

将指定的符号 *name* 添加到专用符号表。MDB 提供了一个可以用于插入目标符号表的专用可配置符号表，如第 24 页中的 “符号名称解析”中所述。`::nmadd` dcmd 还可识别以下选项：

`-e`	将符号大小设置为 *end* - *value*。
`-f`	将符号类型设置为 `STT_FUNC`。
`-o`	将符号类型设置为 `STT_OBJECT`。
`-s`	将符号大小设置为 *size*。

`::nmdel` *name*

从专用符号表中删除指定的符号 *name*。

`::objects` [`-v`]

列显目标虚拟地址空间的映射，仅显示与每个已知装入对象的专用映射（通常为文本部分）相对应的那些映射。如果存在 `-v` 选项，则在知道版本信息的情况下，该命令显示每个对象的版本。如果不知道版本信息，则将在输出中显示 `Unknown` 版本。

`::offsetof` *type member*

列显指定类型的指定成员的偏移。类型应该为 C 结构的名称。除非成员是可以按位列显偏移的位字段，否则按字节列显偏移。为清楚起见，输出始终使用相应单位作为后缀。类型名称可以使用第 24 页中的 “符号名称解析”中所述的反引号 (‘) 作用域运算符。

注 – 此 dcmd 只能与包含压缩的符号调试信息的对象（设计用于 `mdb`）一起使用。此信息当前仅可用于某些 Solaris 内核模块。必须安装 SUNWzlib 解压缩软件，才能处理符号调试信息。

[*address* [, *len*]] `::out` [`-L` *len*]

将指定的 *value* 写入 *address* 指定的 I/O 端口。如果存在 `-L` 选项的值，则它优先于在左侧指定的重复计数。*len* 必须为 1、2 或 4 字节，而且端口地址必须按照长度对齐。仅当在 x86 系统上使用 `kmdb` 时，此命令才可用。

[*address*] `::print [-aCdiLptx] [-c` *lim*`] [-l` *lim*`] [`*type* `[`*member*`|`*offset* `... ]]`

使用给定的 *type* 信息，列显指定的虚拟 *address* 上的数据结构。*type* 参数可以指定 C 结构、联合、枚举、基础整数类型或指向这些类型之一的指针。如果类型名称包含空格（例如 "struct foo"），则必须用单引号或双引号将它引起来。类型名称可以使用第 24 页中的 “符号名称解析”下所述的反引号 (‘) 作用域运算符。如果类型为结构化类型，则 `::print` dcmd 将以递归方式列显结构或联合的每个成员。如果 *type* 参数不存在，但静态或全局 STT_OBJECT 符号与地址匹配，则 `::print` 将自动推断出相应的类型。

type 参数可以后跟 *member* 或 *offset* 表达式的可选列表，在这种情况下仅显示指定的 *type* 的那些成员和子成员。可以使用包括数组索引运算符 (`[]`)、结构成员运算符 (`->`) 和结构指针运算符 (`.`) 的 C 语法指定成员。可以使用 MDB 算术扩展语法 (`$[]`) 指定偏移。显示数据结构后，`::print` 将按 *type* 的大小（以字节为单位）递增点。

注 – `::print` dcmd 只能与包含压缩的符号调试信息的对象（设计用于 MDB）一起使用。目前，此信息仅在某些 Solaris 内核模块和用户库中可用。必须安装 `SUNWzlib` 解压缩软件，才能处理符号调试信息。

如果存在 `-a` 选项，则显示每个成员的地址。如果存在 `-i` 选项，则将左侧的表达式解释为要使用指定类型显示的即时值。如果存在 `-p` 选项，则 `::print` 将地址解释为物理内存地址而不是虚拟内存地址。如果存在 `-t` 选项，则显示每个成员的类型。如果存在 `-d` 或 `-x` 选项，则以十进制 (`-d`) 或十六进制 (`-x`) 显示所有整数；缺省情况下，使用试探性算法确定应该以十进制还是十六进制显示值。可以使用 `-c` 选项限制字符数组中将读取并显示为字符串的字符数。如果存在 `-C` 选项，则不实施任何限制。可以使用 `-l` 选项限制标准数组中将读取并显示的元素数。如果存在 `-L` 选项，则不实施任何限制，将显示所有的数组元素。可以使用 `::set` 或 `-o` 命令行选项修改 `-c` 和 `-l` 的缺省值，如附录 A 中所述。

`::quit` [`-u`]
`$q` [`-u`]
退出调试器。`-u` 选项导致调试器恢复执行操作系统并卸载调试器（仅在使用 `kmdb` 时可用）。如果在引导时装入了 `kmdb`，则无法使用 `-u` 选项。如果 `-u` 选项不存在，则 `::quit` 导致 `kmdb` 退出到固件（在 SPARC 系统上）或导致系统重新引导（在 x86 系统上）。

[*thread*] `::regs`
[*thread*] `$r`
列显代表线程的典型寄存器集。如果指定了一个线程，则显示该线程的典型寄存器集。线程表达式应该为第 48 页中的 “线程支持”下说明的线程标识符之一。

`::release` [`-a`]
`:R` [`-a`]
释放以前附加的进程或核心转储文件。如果存在 `-a` 选项，则释放进程，并使其保持已停止和已放弃状态。随后可以由 `prun(1)` 继续执行它，或者通过应用 MDB 或其他调试器来恢复它。缺省情况下，如果释放的进程是 MDB 使用 `::run` 创建的，则它将被强行终止；如果它是 MDB 使用 `-p` 选项或者使用 `::attach` 或 `:A` dcmd 附加的，则它将被释放并设置为正在运行。

`::set` [`-wF`] [`+/-o` *option*] [`-s` *distance*] [`-I` *path*] [`-L` *path*] [`-P` *prompt*]
获取或设置各种调试器属性。如果未指定选项，则显示当前的调试器属性集。`::set` dcmd 可识别以下选项：

`-F` 强制接管对其应用 `::attach` 的下一个用户进程，就像已在命令行上使用 `-F` 选项执行了 `mdb`。

`-I` 设置用于查找宏文件的缺省路径。路径参数可以包含在附录 A 中为 `-I` 命令行选项说明的任何特殊标记。

`-L` 设置用于查找调试器模块的缺省路径。路径参数可以包含在附录 A 中为 `-I` 命令行选项说明的任何特殊标记。

-o 启用指定的调试器选项。 如果使用 +o 形式，则禁用该选项。 选项字符串以及 -o 命令行选项在附录 A 中说明。

-P 将命令提示符设置为指定的提示字符串。

-s 将符号匹配距离设置为指定的距离。有关更多信息，请参阅附录 A 中对 -s 命令行选项的说明。

-w 重新打开目标以写入，就像已在命令行上使用 -w 选项执行了 mdb。

::showrev [-pv]
显示与当前目标对应的硬件和软件修订信息。如果未指定选项，则显示常规系统信息。如果存在 -p 选项，则显示作为修补程序一部分的每个装入对象的信息。如果存在 -v 选项，则显示每个装入对象的信息。-p 选项的输出将省略没有版本信息的装入对象。在 -v 选项的输出中，没有版本信息的装入对象将报告 Unknown。

::sizeof *type*
列显指定类型的大小（以字节为单位）。*type* 参数可以指定 C 结构、联合、枚举、基础整数类型或指向这些类型之一的指针。类型名称可以使用第 24 页中的 “符号名称解析” 中所述的反引号 (‘) 作用域运算符。

注 – 此 dcmd 只能与包含压缩的符号调试信息的对象（设计用于 mdb）一起使用。此信息当前仅可用于某些 Solaris 内核模块。必须安装 SUNWzlib 解压缩软件，才能处理符号调试信息。

[*address*] ::stack [*count*]
[*address*] $c [*count*]
列显 C 栈反向跟踪。如果 dcmd 的前面有显式的 *address*，则显示从此虚拟内存地址开始的反向跟踪。否则，显示代表线程的栈。如果提供了可选计数值作为参数，则在输出中为每个栈帧显示的参数不超过 *count* 个。

仅适用于 64-bit SPARC – 请求栈跟踪时，应该将偏离的帧指针值（即，虚拟地址减去 0x7ff）用作地址。

::status
列显与当前目标相关的信息的摘要。

cpuid ::switch
cpuid :x
切换到由指定的 *cpuid* 所指示的 CPU，并将此 CPU 的当前寄存器状态用作代表以进行调试（仅在使用 kmdb 时可用）。

::term
列显 MDB 用来执行任何与终端相关的输入和输出操作（如命令行编辑）的终端类型的名称。

thread ::tls *symbol*
列显指定线程上下文中指定的线程局部存储 (thread-local storage, TLS) 符号的存储地址。线程表达式应该为第 48 页中的 “线程支持” 下说明的线程标识符之一。符号名称可以使用第 24 页中的 “符号名称解析” 下说明的任何作用域运算符。

`::typeset [+/-t]` *variable-name* ...
设置命名变量的属性。如果指定了一个或多个变量名称，则将定义它们并将其设置为点的值。如果存在 `-t` 选项，则设置与每个变量关联的用户定义标记。如果存在 `+t` 选项，则清除标记。如果未指定变量名称，则列显变量及其值的列表。

`::unload` *module-name*
卸载指定的 dmod。可以使用 `::dmods` dcmd 列显活动 dmod 的列表。无法卸载内置模块。无法卸载处于忙状态（即，提供当前正在执行的 dcmd）的模块。

`::unset` *variable-name* ...
从已定义变量列表中取消设置（删除）指定的变量。由 MDB 导出的一些变量标记为持久性变量，用户无法取消设置它们。

`::vars [-npt]`
列显命名变量的列表。如果存在 `-n` 选项，则将输出限制为当前具有非零值的变量。如果存在 `-p` 选项，则以适合于调试器使用 `$<` dcmd 重新处理的形式列显变量。此选项可以用于将变量记录到宏文件，稍后再恢复这些值。如果存在 `-t` 选项，则仅列显带标记的变量。可以使用 `::typeset` 的 `-t` 选项标记变量。

`::version`
列显调试器版本号。

address `::vtop [- a` *as*`]`
如有可能，则列显指定虚拟地址的物理地址映射。仅在检查内核目标或检查内核崩溃转储内的用户进程（发出 `::context` dcmd 之后）时， `::vtop` dcmd 才可用。

从内核上下文检查内核目标时，可以使用 `-a` 选项指定应该用于从虚拟到物理转换的替换地址空间结构的地址 (*as*)。缺省情况下，使用内核的地址空间进行转换。 此选项可用于活动地址空间，甚至是在转储内容仅包含内核页面时，也是如此。

[*address* } `::walk` *walker-name* [*variable-name*]
使用指定的 walker 遍历数据结构的元素。可以使用 `::walkers` dcmd 列出可用的 walker。一些 walker 对全局数据结构执行操作，并且不需要起始地址。例如，遍历内核中 `proc` 结构的列表。其他 walker 对必须显式指定其地址的特定数据结构执行操作。例如，在提供指向地址空间的指针时，遍历段的列表。

以交互方式使用时， `::walk` dcmd 将以缺省基数列显每个数据结构元素的地址。该 dcmd 还可以用于为管道提供地址列表。walker 名称可以使用第 25 页中的 “dcmd 和 Walker 名称解析”中说明的反引号 "‘" 作用域运算符。如果指定了可选的 *variable-name*，则 MDB 调用管道的下一个阶段时， 会将在遍历的每一步返回的值赋给指定的变量。

`::walkers`
列出可用的 walker 并列显其中每个 walker 的简短说明。

`::whence [-v]` *name* ...
`::which [-v]` *name* ...
列显导出指定的 dcmd 和 walker 的 dmod。这些 dcmd 可以用于确定当前哪个 dmod 提供给定的 dcmd 或 walker 的全局定义。有关全局名称解析的更多信息，请参阅第 25 页中的 “dcmd 和 Walker 名称解析” 。`-v` 选项导致 dcmd 按优先级顺序列显每个 dcmd 和 walker 的替换定义。

`::xdata`

列出当前目标所导出的外部数据缓冲区。外部数据缓冲区表示与无法通过标准目标工具（即，地址空间、符号表或寄存器集）访问的目标关联的信息。这些缓冲区可以由 dcmd 占用；有关更多信息，请参阅第 126 页中的 “mdb_get_xdata()”。

第 6 章

执行控制

MDB 提供了用于控制和跟踪实时运行程序（包括用户应用程序以及实时操作系统内核和设备驱动程序）执行的工具。可以使用 mdb 命令控制已运行的用户进程或者在调试器的控制下创建新进程。可以引导或装入 kmdb 以控制操作系统内核本身的执行或者调试设备驱动程序。本章介绍可以用于控制目标执行的内置 dcmd。除非另行说明，否则可以在 mdb 或 kmdb 中使用这些命令。仅与 kmdb 中执行控制相关的其他主题将在第 7 章中讨论。

执行控制

MDB 提供了一种简单的执行控制模型：可以使用 ::run 从调试器内启动目标进程，或者MDB 可以使用 :A、::attach 或 -p 命令行选项（请参见第 5 章）附加到现有进程。或者，可以使用 kmdb 引导内核，也可以在以后装入 kmdb。在任一情况下，用户都可以指定跟踪的**软件事件**的列表。每次跟踪的事件在目标程序中发生时，目标中的所有线程都会停止，触发该事件的线程会被选为代表线程，并且控制返回到调试器。将目标程序设置为运行后，即可通过键入用户定义的中断字符（通常为 Ctrl-C）将控制以异步方式返回到调试器。

软件事件是指调试器所观察到的目标程序中的状态转换。例如，调试器可能会观察到程序计数器寄存器到相关值（断点）的转换或特定信号的传送。

软件事件说明符是指调试器所使用的软件事件类的说明，用于检测目标程序以便观察这些事件。 ::events dcmd 用于列出软件事件说明符。 一组标准属性与每个事件说明符相关联，如第 49 页中的 “内置 dcmd”中 ::events 之下所述。

调试器可以观察各种不同的软件事件，包括断点、监视点、信号、计算机故障和系统调用。 使用 ::bp、::fltbp、:: sigbp、::sysbp 或 ::wp 可以创建新说明符。每个说明符都具有关联的回调（要执行的 MDB 命令字符串，就好像已在命令提示符下键入了它一样）和一组属性，如第 49 页中的 “内置 dcmd”中 ::events 之下所述。可以为同一事件创建任意数量的说明符，每个说明符都具有不同的回调和属性。使用 ::events dcmd 可以显示跟踪的事件和对应事件说明符的属性的当前列表。 事件说明符属性是作为第 49 页中的 “内置 dcmd”中 ::events 和 ::evset dcmd 说明的一部分定义的。

第 49 页中的 “内置 dcmd”中所述的执行控制的内置 dcmd 始终可用，但是如果将其应用于不支持执行控制的目标，则会发出一条错误消息，表明其不受支持。

事件回调

通过 ::evset dcmd 和用于事件跟踪的 dcmd，可以将事件回调与每个事件说明符关联（使用 -c 选项）。事件回调是指用于表示在目标中发生对应事件时要执行的 MDB 命令的字符串。系统将执行这些命令，就好像已在命令提示符下键入了它们一样。执行每个回调之前，会将 *dot* 变量设置为代表线程的程序计数器的值，并且将 *hits* 变量设置为与此说明符匹配的次数（包括当前匹配）。

如果事件回调本身包含一个或多个用于继续执行目标的命令（例如 ::cont 或 ::step），则这些命令**不会**立即继续执行目标并等待其再次停止。相反，在某个事件回调内，用于继续执行的 dcmd 会注意到继续操作现在处于暂挂状态，然后会立即返回。因此，如果某个事件回调中包含多个 dcmd，则用于单步执行或继续执行的 dcmd 应该是指定的最后一个命令。执行**所有**事件回调后，如果**所有**匹配的事件回调都请求继续操作，则目标将立即恢复执行。如果请求的继续操作相冲突，则优先级最高的操作将确定发生的继续类型。优先级从高到低的顺序如下：单步执行、步过（下一个）、步出、继续。

线程支持

MDB 提供了用于检查与目标关联的每个线程的栈和寄存器的工具。持久性 "thread" 变量包含当前代表线程的标识符。线程标识符的格式取决于目标。::regs 和 ::fpregs dcmd 可以用于检查代表线程的寄存器集或其他线程的寄存器集（如果其寄存器集当前可用）。此外，代表线程的寄存器集会作为一组命名变量导出。 用户可以通过将 > dcmd 应用于对应的命名变量来修改一个或多个寄存器的值。

MDB 内核目标会将对应的内部线程结构的虚拟地址导出作为指定线程的标识符。此地址与操作系统源代码中的 kthread_t 数据结构相对应。使用 kmdb 时，运行 kmdb 的 CPU 的 CPU 标识符会存储在 cpuid 变量中。

MDB 进程目标为检查使用本机 lwp_* 接口、/usr/lib/libthread.so 或 /usr/lib/libpthread.so 的多线程用户进程提供了适当的支持。调试实时用户进程时，MDB 将检测单线程进程是通过 dlopen 打开还是关闭 libthread，并将自动调整其所使用的线程模型视图。 进程目标线程标识符将与代表线程的 lwpid_t、thread_t 或 pthread_t 对应，具体取决于应用程序所使用的线程模型。

如果 MDB 正在调试用户进程目标，并且目标可利用编译器支持的线程局部存储空间，则 MDB 会自动将引用线程局部存储空间的符号名称评估为对应于当前代表线程的存储空间地址。::tls 内置 dcmd 可以用于显示除代表线程之外的线程的符号值。

内置 dcmd

[*addr*] ::bp [+/-dDestT] [-c *cmd*] [-n *count*] *sym* ...
addr :b [*cmd* ...]

用于在指定的位置设置断点。 ::bp dcmd 可在指定的每个地址或符号上设置断点，其中包括 dcmd 前面的显式表达式所指定的可选地址，以及 dcmd 后面的每个字符串或即时值。参数可以是表示特定的相关虚拟地址的符号名称或即时值。如果指定了符号名称，则它可能引用还无法在目标进程中评估的符号：即该符号可能包含尚未打开的装入对象中的对象名称和函数名称。 在这种情况下，断点将会延迟，在装入与指定名称匹配的对象之前，它在目标中不会处于活动状态。打开装入对象时，将自动启用断点。共享库中所定义的符号上的断点应该始终使用符号名称而不是地址表达式进行设置，因为地址可能会引用对应的过程链接表 (Procedure Linkage Table, PLT) 项而不是实际的符号定义。如果随后将 PLT 项解析为实际的符号定义，则运行时链接编辑器可能会覆写在该 PLT 项上设置的断点。如本节中稍后所述，-d、-D、-e、-s、-t、-T、-c 和 -n 选项具有与用于 ::evset dcmd 时相同的含义。如果使用的是 dcmd 的 :b 形式，则仅会在 dcmd 前面的表达式所指定的虚拟地址上设置断点。 :b dcmd 后面的参数会串联在一起形成回调字符串。 如果此字符串包含元字符，则必须使用引号将此字符串引起来。

function ::call [*arg* ...]

调用操作系统内核中所定义的指定 *function*（仅在使用 kmdb 时可用）。*function* 表达式必须与其中一个已知内核模块的符号表中所定义的函数的地址匹配。如果指定的是表达式参数，则这些参数将通过值进行传递。如果指定的是字符串参数，则这些参数将通过引用进行传递。

注 – 请务必谨慎使用 ::call 命令，决不可将其应用于产品化系统。操作系统内核不会为了执行指定的函数而恢复执行。因此，被调用的函数决不能利用任意内核服务，并且决不能由于任何原因而阻塞。必须完全了解使用此命令调用的任何函数的副作用。

::cont [*SIG*]
:c [*SIG*]

用于暂停调试器，继续执行目标程序，并等待它在相关软件事件发生后终止或停止。如果由于在启用了 -o nostop 选项的情况下将调试器附加到正在运行的程序，从而导致目标已在运行，则此 dcmd 仅等待目标在相关事件发生后终止或停止。如果将可选的信号名称或编号（请参见 signal(3HEAD) 手册页）指定为参数，则信号会在其恢复执行的过程中立即传送到目标。如果跟踪了 SIGINT 信号，则可通过键入用户定义的中断字符（通常为 ^C）将控制以异步方式返回到调试器。 此 SIGINT 信号会自动清除，下次继续执行此信号时目标不会观察到它。 如果当前没有任何目标程序正在运行，则 ::cont 将开始运行一个新程序，就好像通过 ::run 执行一样。

addr ::delete [*id* | all]
addr :d [*id* | all]

用于删除具有指定 id 号的事件说明符。缺省情况下，*id* 编号参数按十进制解释。 如果在 dcmd 前面指定了可选地址，则将删除与指定虚拟地址关联的所有事件说明符（例如，影响该地址的所有断点或监视点）。如果指定了特殊参数 "all"，则将删除所有事件说明符，但标记为粘滞（T 标志）的说明符除外。 ::events dcmd 可用于显示事件说明符的当前列表。

::events [-av]

$b [-av]

用于显示软件事件说明符的列表。 针对每个事件说明符都会指定一个唯一的 ID 号，稍后可以使用该编号对其进行删除或修改。调试器可能还会启用自己的内部事件进行跟踪；仅当存在 -a 选项时，才会显示这些内部事件。 如果存在 -v 选项，则将显示更详细的内容，其中包括任何说明符不活动的原因。以下的 ::events dcmd 显示了示例输出：

```
> ::events
   ID S TA HT LM Description                              Action
----- - -- -- -- ---------------------------------------- -------------
[ 1 ] - T   1  0 stop on SIGINT                           -
[ 2 ] - T   0  0 stop on SIGQUIT                          -
[ 3 ] - T   0  0 stop on SIGILL                           -
 ...
[ 11] - T   0  0 stop on SIGXCPU                          -
[ 12] - T   0  0 stop on SIGXFSZ                          -
[ 13] -     2  0 stop at libc‘printf                      ::echo printf
>
```

以下讨论说明了每列的含义。使用 ::help events 可以显示此信息的摘要。

ID 事件说明符的标识符。如果启用了说明符，则将在方括号 [ ] 中显示标识符；如果禁用了说明符，则在圆括号 ( ) 中显示标识符；如果当前在与指定说明符匹配的事件上停止了目标程序，则在尖括号 < > 中显示标识符。

S 事件说明符的状态。 状态将是以下符号之一：

-	事件说明符处于空闲状态。如果没有任何目标程序运行，则表明所有说明符都处于空闲状态。目标程序正在运行时，如果无法评估说明符，则表明该说明符可能处于空闲状态（如尚未装入的共享对象中的延迟断点）。
+	事件说明符处于活动状态。继续执行目标时，调试器将检测到此类型的事件。
*	事件说明符处于待命状态。此状态意味着目标当前正在运行，并且可检测到此类型的事件。仅当使用 -o nostop 选项将调试器附加到正在运行的程序时，此状态才可见。

!	由于出现操作系统错误，事件说明符未处于待命状态。 可以使用 ::events -v 选项显示有关检测失败原因的更多信息。

TA　“临时”、“粘滞”和“自动”事件说明符的属性。可能显示以下符号中的一个或多个：

t	事件说明符是临时的，无论它是否匹配，下次目标停止时都会将其删除。
T	事件说明符是粘滞说明符，不会被 ::delete all 或 :z 删除。通过将其 id 号显式指定为 ::delete，可以删除该说明符。
d	命中计数等于命中限制时，将自动禁用事件说明符。
D	命中计数等于命中限制时，将自动删除事件说明符。
s	命中计数等于命中限制时，目标将自动停止。

HT　当前的命中计数。 此列显示自创建此事件说明符以来对应软件事件在目标中发生的次数。

LM　当前的命中限制。此列显示自动禁用、自动删除或自动停止行为将生效的命中计数限制。可以使用 ::evset dcmd 配置这些行为。

Description　对给定的说明符所匹配的软件事件类型的说明。

Action　对应软件事件发生时要执行的回调字符串。系统将执行此回调，就好像已在命令提示符下键入了它一样。

id ::evset [+/-dDestT] [-c *cmd*] [-n *count*] *id* ...

用于修改一个或多个软件事件说明符的属性。将设置由 dcmd 前面的可选表达式和 dcmd 后面的可选参数列表所标识的每个说明符的属性。除非指定显式基数，否则参数列表将被解释为十进制整数的列表。::evset dcmd 可识别以下选项：

-d　当命中计数达到命中限制时，禁用事件说明符。如果指定了该选项的 +d 形式，则会禁用此行为。禁用事件说明符后，调试器便会删除任何对应检测并忽略对应的软件事件，直到随后重新启用说明符为止。 如果 -n 选项不存在，则会立即禁用说明符。

-D　当命中计数达到命中限制时，删除事件说明符。如果指定了该选项的 +D 形式，则会禁用此行为。-D 选项优先于 -d 选项。可以使用 -n 选项配置命中限制。

-e　启用事件说明符。如果指定了该选项的 +e 形式，则会禁用说明符。

-s　当命中计数达到命中限制时，停止目标程序。如果指定了该选项的 +s 形式，则会禁用此行为。 -s 行为通知调试器就像在每次执行说明符回调后发出 ::cont 那样进行操作，但第 N 次执行除外，其中 N 是说明符命中限制的当前值。-s 选项优先于 -D 选项和 -d 选项。

- `-t` 将事件说明符标记为临时说明符。无论临时说明符是否由于对应于指定说明符的软件事件而停止，下次目标停止时都会自动将其删除。如果指定了该选项的 `+t` 形式，则会删除临时标记。`-t` 选项优先于 `-T` 选项。
- `-T` 将事件说明符标记为粘滞。 粘滞说明符不会被 `::delete all` 或 `:z` 删除。通过将对应的说明符 ID 指定为 `::delete` 的显式参数，可以将其删除。如果指定了该选项的 `+T` 形式，则会删除粘滞属性。缺省的一组事件说明符最初都标记为粘滞。
- `-c` 每次目标程序中发生对应软件事件时，执行指定的 *cmd* 字符串。可以使用 `::events` 显示当前的回调字符串。
- `-n` 将命中限制的当前值设置为 *count*。如果当前未设置任何命中限制，并且 `-n` 选项未附带 `-s` 或 `-D`，则命中限制将被设置为 1。

使用 `::help evset` 可以显示此信息的摘要。

flt `::fltbp` [+/-dDestT] [`-c` *cmd*] [`-n` *count*] *flt* ...
用于跟踪指定的计算机故障。使用 dcmd 前面的可选故障编号或者 dcmd 后面的故障名称或编号的列表（请参见 `<sys/fault.h>`）来标识故障。`-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c` 和 `-n` 选项具有与用于 `::evset` dcmd 时相同的含义。`::fltbp` 命令仅适用于用户进程调试。

signal `:i`
如果目标是实时用户进程，请忽略指定的信号，并允许以透明方式将其传送到目标。将从跟踪的事件列表中删除跟踪指定信号传送的所有事件说明符。缺省情况下，忽略的信号集会初始化为导致进程缺省情况下转储核心的信号集的补充（请参见 signal(3HEAD) 手册页），但 SIGINT 除外，缺省情况下将跟踪此信号。`:i` 命令仅适用于用户进程调试。

`$i`
显示调试器忽略的将直接由目标处理的信号的列表。使用 `::events` dcmd 可以获取有关跟踪的信号的更多信息。`$i` 命令仅适用于用户进程调试。

`::kill`
`:k`
如果目标是实时用户进程，则强制将其终止。如果目标是调试器使用 `::run` 创建的，则在调试器退出时，也将强制终止该目标。`::kill` 命令仅适用于用户进程调试。

`$l`
如果目标是用户进程，则列显代表线程的 LWPID。

`$L`
如果目标是用户进程，则列显该目标中每个 LWP 的 LWPID。

`::next` [*SIG*]
`:e` [*SIG*]
一次执行目标程序的一条指令，但步过子例程调用。如果将可选的信号名称或编号（请参见 signal(3HEAD) 手册页）指定为参数，则信号会在其恢复执行的过程中立即传送到目标。如果当前没有任何目标程序正在运行，则 `::next` 将开始运行新程序，就好像是通过 `::run` 执行一样，然后在第一条指令处停止。

`::run` [*args* ...]
`:r` [*args* ...]

用于通过指定的参数开始运行新的目标程序，并附加到该程序。这些参数不是通过 shell 解释的。如果调试器已经在检查实时运行的程序，则它将首先与该程序分离，就好像是通过 `::release` 执行一样。

[*signal*] `::sigbp` [+/-dDestT] [`-c` *cmd*] [`-n` *count*] *SIG* ...
[*signal*] `:t` [+/-dDestT] [`-c` *cmd*] [`-n` *count*] *SIG* ...

用于跟踪指定信号的传送。 使用 dcmd 前面的可选信号编号或者 dcmd 后面的信号名称或编号的列表（请参见 signal(3HEAD)）来标识信号。 `-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c` 和 `-n` 选项具有与用于 `::evset` dcmd 时相同的含义。 最初会跟踪缺省情况下导致进程转储核心的信号集（请参见 signal(3HEAD)）和 SIGINT。`::sigbp` 命令仅适用于用户进程调试。

`::step` [`branch` | `over` | `out`] [*SIG*]
`:s` *SIG*
`:u` *SIG*

用于执行目标程序的一条指令。如果将可选的信号名称或编号（请参见 signal(3HEAD) 手册页）指定为参数并且目标是用户进程，则信号会在其恢复执行的过程中立即传送到目标。如果指定了可选的 `branch` 参数，则在用于对处理器控制流进行分支的下一条指令之前，目标程序将继续执行。仅当针对启用了相应处理器特定功能的 x86 系统使用 `kmdb` 时，`::step branch` 功能才可用。如果指定了可选的 `over` 参数，则 `::step` 将跳过子例程调用。 `::step over` 参数与 `::next` dcmd 相同。如果指定了可选的 `out` 参数，则在代表线程从当前函数返回之前，目标程序将继续执行。如果当前没有任何目标程序正在运行，则 `::step over` 将开始运行新程序，就好像是通过 `::run` 执行一样，然后在第一条指令处停止。`:s` dcmd 与 `::step` 相同。`:u` dcmd 与 `::step out` 相同。

[*syscall*] `::sysbp` [+/-dDestT] [`-io`] [`-c` *cmd*] [`-n` *count*] *syscall* ...

用于跟踪指定系统调用的进出信息。 使用 dcmd 前面的可选系统调用编号或者 dcmd 后面的系统调用名称或编号的列表（请参见 `<sys/syscall.h>`）来标识系统调用。 如果指定了 `-i` 选项（缺省值），则事件说明符会在每个系统调用进入内核时触发。 如果指定了 `-o` 选项，则事件说明符会在从内核退出时触发。`-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c` 和 `-n` 选项具有与用于 `::evset` dcmd 时相同的含义。`::sysbp` 命令仅适用于用户进程调试。

addr [,*len*] `::wp` [+/-dDestT] [`-rwx`] [`-ip`] [`-c` *cmd*] [`-n` *count*]
addr [,*len*]`:a` [*cmd*...]
addr [,*len*]`:p` [*cmd*...]
addr [,*len*] `:w` [*cmd*...]

用于在指定的地址上设置监视点。通过在 dcmd 前面指定可选的重复计数，可以设置监视的区域的长度（以字节为单位）。如果未显式设置长度，则缺省长度为 1 个字节。使用 `::wp` dcmd 可将监视点配置为由读取（`-r` 选项）、写入（`-w` 选项）或执行（`-x` 选项）访问的任意组合操作触发。 `-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c` 和 `-n` 选项具有与用于 `::evset` dcmd 时相同的含义。可以使用 `-i` 选项指明应该在 I/O 端口的地址上设置监视点（仅当在 x86 系统上使用 `kmdb` 时可用）。可以使用 `-p` 选项指明应该将指定的地址解释为物理地址（仅在使用 `kmdb` 时可用）。`:a` dcmd 用于在指定的地址上设置读取访问监视点。`:p` dcmd 用于在指定的地址上设置执行访问监视点。 `:w` dcmd 用于在指定的地址上设置写入访问监视点。 `:a`、`:p` 和 `:w` dcmd 后面的参数会串联在一起形成回调字符串。 如果此字符串包含元字符，则必须使用引号将此字符串引起来。

`:z`

用于从跟踪的软件事件列表中删除所有事件说明符。另外，也可以使用 `::delete` 删除事件说明符。

与 exec 交互

如果受控的用户进程成功执行 exec(2)，则可通过 ::set -o follow_exec_mode 选项控制调试器的行为，如第 127 页中的 “命令行选项摘要”中所述。如果调试器和被调试的进程 (victim process) 具有相同的数据模型，则 “停止” 和 “跟随” 模式可确定 MDB 是自动继续执行目标还是返回到 exec 后面的调试器提示符下。如果调试器和被调试的进程具有不同的数据模型，则 “跟随” 行为导致 MDB 使用相应的数据模型自动地重新执行 MDB 二进制代码并重新附加到该进程，从 exec 返回时仍然被停止。 并非所有在此重新执行中生成的调试器状态都会保留。

如果在 32 位数据模型中被调试的进程对 64 位程序执行 exec，则 MDB 在 “停止” 模式下将返回到命令提示符，但调试器将无法再检查进程，因为现在它使用的是 64 位数据模型。要恢复调试，请执行 ::release -a dcmd，退出 MDB，然后执行 mdb -p *pid* 将 64 位调试器重新附加到进程。

如果在 64 位数据模型中被调试的进程执行 32 位程序，则命令提示符中将返回 “停止”，但调试器将仅提供有限的功能用于检查新进程。 所有内置 dcmd 都将按照说明的那样运行，但是可装入的 dcmd 则不会，因为它们不会执行结构的数据模型转换。 如前所述，为了恢复完整的调试功能，用户应该释放调试器再将其重新附加到进程。

与作业控制交互

如果将调试器附加到作业控制所停止的用户进程（即，该用户进程响应 SIGTSTP、SIGTTIN 或 SIGTTOU 而停止），则用于继续执行操作的 dcmd 继续执行进程时，可能无法将该进程设置为再次运行。 如果被调试的进程是同一会话的成员（即，该进程与 MDB 共享同一控制终端），则 MDB 会尝试将关联的进程组置于前台，并响应 SIGCONT 继续执行进程，以便将其从作业控制停止中恢复。 MDB 与此类进程分离时，它在退出之前会将进程组恢复到后台。如果被调试的进程不是同一会话的成员，则 MDB 无法安全地将进程组置于前台，因此它将继续执行与调试器有关的进程，但作业控制会继续停止进程。在这种情况下，MDB 将列显一条警告，用户必须从相应的 shell 发出 fg 命令才能恢复进程。

进程的附加和释放

如果 MDB 附加到正在运行的用户进程，则会停止该进程并使其一直处于停止状态，直到应用一个用于继续执行操作的 dcmd 或调试器退出为止。 如果在使用 -p 将调试器附加到某个进程之前或者发出 ::attach 或 :A 命令之前启用了 -o nostop 选项，MDB 将会附加到该进程但不会将其停止。在进程仍然运行的同时，可以按通常那样对其进行检查（尽管结果会不一致），并且可以启用断点或其他跟踪标志。 如果在进程运行的同时执行 :c 或 ::cont dcmd，则调试器将等待进程停止。如果没有发生跟踪的软件事件，则用户可以在 :c 或 ::cont 之后发送中断字符 (^C)，以强制进程停止并将控制返回到调试器。

执行 :R、::release、:r、::run、$q 或 ::quit dcmd 时，或者调试器因遇到 EOF 或信号而终止时，MDB 将释放当前运行的进程（如果有）。如果进程最初是调试器使用 :r 或 ::run 创建的，则在释放该进程时会强制将其终止，就好像是通过 SIGKILL 执行一样。 如果在将

MDB 附加到进程之前该进程已经运行，则在释放该进程时会将其设置为再次运行。使用 `::release -a` 选项，可以释放进程并使其保持停止和放弃状态。

◆ ◆ ◆ 第 7 章

内核执行控制

本章介绍运行 kmdb 时可用于实时操作系统内核执行控制的 MDB 功能。kmdb 是 MDB 的一种版本，专为内核执行控制和实时内核调试而设计。 使用 kmdb，可以与使用 mdb 控制和观察用户进程几乎相同的方式控制和观察内核。 内核执行控制功能包括每个 CPU 上执行的内核线程的指令级控制，这样开发者就可以单步控制内核以及实时检查数据结构。

mdb 和 kmdb 共享相同的用户界面。第 6 章中介绍的所有执行控制功能在 kmdb 中都可用，它们与用于控制用户进程的一组命令相同。 第 3 章和第 5 章中介绍的用于检查内核状态的命令在使用 kmdb 时也是可用的。最后，除非另有说明，否则第 8 章中介绍的特定于 Solaris 内核实现的命令也可用。本章介绍特定于 kmdb 的其他功能。

引导、装入和卸载

为便于启动内核调试，可以在控制从内核运行时链接程序 (krtld) 传递到内核之前，在引导过程的初期阶段装入 kmdb。可以使用 -k 引导标志、kmdb 引导文件或 kadb 引导文件（用于实现兼容性）在引导时装入 kmdb。如果在引导时装入 kmdb，则在系统随后重新引导之前，无法卸载调试器。 在引导的初期阶段，某些功能不会立即可用。 特别是，在内核模块子系统初始化之前，不会装入调试模块。 在内核完成处理器标识过程之前，不会启用处理器特定功能。

如果使用 -k 选项引导系统，则会在引导过程中自动装入 kmdb。 可以使用 -d 引导选项，在启动内核之前请求调试器断点。 此功能适用于缺省内核以及替代内核。例如，要使用 kmdb 引导 SPARC 系统并请求立即进入调试器，请键入以下任一命令：

```
ok boot -kd
```

```
ok boot kmdb -d
```

```
ok boot kadb -d
```

要以相同方式引导 x86 系统，请键入以下任一命令：

```
Select (b)oot or (i)nterpreter: b -kd
```

```
Select (b)oot or (i)nterpreter: b kmdb -d
```

```
Select (b)oot or (i)nterpreter: b kadb -d
```

要使用 kmdb 引导 SPARC 系统并装入备用 64 位内核，请键入以下命令：

```
ok boot kernel.test/sparcv9/unix -k
```

要使用 kmdb 引导 x86 系统并装入备用 64 位内核，请键入以下命令：

```
Select (b)oot or (i)nterpreter: b kernel.test/amd64/unix -k
```

如果引导文件被设置为字符串 kmdb 或 kadb，并且希望引导替代内核，请使用 -D 选项指定要引导的内核的名称。要以此方式引导 SPARC 系统，请键入以下命令：

```
ok boot kmdb -D kernel.test/sparcv9/unix
```

要以此方式引导 32 位 x86 系统，请键入以下命令：

```
Select (b) or (i)nterpreter: b kmdb -D kernel.test/unix
```

要以此方式引导 64 位 x86 系统，请键入以下命令：

```
Select (b) or (i)nterpreter: b kmdb -D kernel.test/amd64/unix
```

要调试已引导的系统，请使用 mdb -K 选项装入 kmdb 并停止内核执行。如果使用此方法装入调试器，则随后可以卸载它。通过为 ::quit dcmd 指定 -u 选项，可以在完成调试后卸载 kmdb。或者，可以使用命令 mdb -U 恢复操作系统的执行。

终端处理

kmdb 始终使用系统控制台进行交互。kmdb 将根据以下规则确定相应的终端类型：

- 如果被调试的系统将连接的键盘和显示器用于其控制台，并且调试器是在引导时装入的，则会根据平台体系结构和控制台终端设置自动确定终端类型。
- 如果被调试的系统使用串行控制台，并且调试器是在引导时装入的，将假定为缺省终端类型 vt100。
- 如果调试器是通过在控制台上运行 mdb -K 装入的，则将 $TERM 环境变量的值用作终端类型。

- 如果调试器是通过在非控制台的终端上运行 `mdb -K` 装入的，则调试器将使用已配置为用于系统控制台登录提示的终端类型。

可以在 `kmdb` 中使用 `::term` dcmd 来显示终端类型。

调试器项

在到达断点时或者根据第 6 章中所述的其他执行控制设置，操作系统内核将隐式停止执行并进入 `kmdb`。可以使用 `mdb -K` 选项或相应的键盘中断序列来请求显式进入 `kmdb`。在 SPARC 系统控制台上，使用 STOP-A 组合键发送中断信号并进入 `kmdb`。在 x86 系统控制台上，使用 F1–A 组合键发送中断信号并进入 `kmdb`。在 Solaris 系统上，可以使用 `kbd` 命令自定义转义序列。要在具有串行控制台的系统上进入 `kmdb`，请使用相应的串行控制台命令发送中断序列。

处理器特定功能

某些 `kmdb` 功能特定于单独的处理器体系结构。例如，各种 x86 处理器支持硬件分支跟踪功能，而在一些其他处理器体系结构中找不到该功能。对处理器特定功能的访问是通过处理器特定的 dcmd（仅存在于支持它们的系统上）提供的。处理器特定支持是否可用将在 `::status` dcmd 的输出中指示。调试器依赖于内核来确定处理器类型。因此，尽管调试器可能为给定的处理器体系结构提供功能，但是在内核前进到已完成处理器标识的点之前，不会公开此支持。

第 8 章

内核调试模块

本章介绍所提供的用于调试 Solaris 内核的调试器模块、dcmd 和 walker。 每个内核调试器模块都以对应的 Solaris 内核模块命名，以便 MDB 可将其自动装入。 此处所述的工具反映当前的 Solaris 内核实现，在将来可能会进行更改；建议不要编写与这些命令的输出相关的 shell 脚本。通常，本章中所介绍的内核调试工具仅在对应的内核子系统实现的上下文中才有意义。有关提供 Solaris 内核实现的更多信息的参考列表，请参见第 10 页中的 "相关书籍和文章"。

注 – 本指南反映 Solaris 10 操作系统实现；由于这些模块、dcmd 和 walker 反映当前的内核实现，因此对于过去或将来的发行版可能并不相关、正确或适用。它们未定义任何类型的永久性公共接口。 在 Solaris 操作环境的未来发行版中，所提供的有关模块、dcmd、walker 及其输出格式和参数的所有信息都可能会进行更改。

通用内核调试支持 (genunix)

内核内存分配器

本节讨论用于调试 Solaris 内核内存分配器所识别的问题以及检查内存和内存使用情况的 dcmd 和 walker。第 9 章中对此处所述的 dcmd 和 walker 进行了更详细的讨论。

dcmd

thread `::allocdby`
: 在指定内核线程的地址的情况下，按反向时间顺序列显其已执行的内存分配的列表。

bufctl `::bufctl` [`-a` *address*] [`-c` *caller*] [`-e` *earliest*] [`-l` *latest*] [`-t` *thread*]
: 列显指定的 bufctl *address* 的 *bufctl* 信息摘要。 如果存在一个或多个选项，则仅当 bufctl 与选项参数所定义的条件匹配时才会列显其信息；这样，可将 dcmd 用作来自管道的输入的过滤器。 `-a` 选项表示 bufctl 的对应缓冲区地址必须等于指定的地址。 `-c` 选项表示指定的调用方的程序计数器值必须存在于 bufctl 的已保存栈跟踪中。 `-e` 选项表示 bufctl 的时

间标记必须晚于或等于指定的最早时间标记。 -l 选项表示 bufctl 的时间标记必须早于或等于指定的最早时间标记。 -t 选项表示 bufctl 的线程指针必须等于指定的线程地址。

[*address*] ::findleaks [-v]
: 对于启用了完整 kmem 调试功能集的内核崩溃转储，::findleaks dcmd 提供了强大高效的内存泄漏检测。 首次执行 ::findleaks 时会处理内存泄漏的转储（这可能需要几分钟），然后按分配栈跟踪合并泄漏。 findleaks 报告显示了所识别的每个内存泄漏的 bufctl 地址和最顶层的栈帧。

  如果指定了 -v 选项，则 dcmd 在执行时会列显更详细的消息。 如果在 dcmd 前面指定了显式地址，则将过滤报告，并仅显示其分配栈跟踪包含指定函数地址的泄漏。

thread ::freedby
: 在指定内核线程的地址的情况下，按反向时间顺序列显其已执行的内存释放的列表。

value ::kgrep
: 在内核地址空间中搜索包含指定的指针大小值的指针对齐地址。 然后，列显包含匹配值的地址的列表。与 MDB 的内置搜索运算符不同，::kgrep 可搜索每段内核地址空间，并跨不连续的段边界搜索。 在大内核上，执行 ::kgrep 可能需要相当长的时间。

::kmalog [slab | fail]
: 显示内核内存分配器的事务日志中的事件。 事件按反向时间顺序显示，首先显示最新的事件。对于每个事件，::kmalog 都显示相对于最新事件的时间（用 T-差值表示法表示，例如 T-0.000151879）、bufctl、缓冲区地址、kmem 高速缓存名称和事件发生时的栈跟踪。 如果没有参数，则 ::kmalog 将显示 kmem 事务日志（仅当 kmem_flags 中设置 KMF_AUDIT 时它才存在）。::kmalog fail 显示分配失败日志，该日志会始终存在；这有助于调试未正确处理分配失败的驱动程序。::kmalog slab 显示长字节创建日志，该日志会始终存在。::kmalog slab 有助于搜索内存泄漏。

::kmastat
: 显示内核内存分配器高速缓存和虚拟内存块 (arena) 的列表，以及对应的统计信息。

::kmausers [-ef] [*cache* ...]
: 列显有关具有当前内存分配的内核内存分配器的中等用户和大用户的信息。对于每个唯一的栈跟踪，输出都包含一个项，用于指定总内存量和在该栈跟踪中进行的分配数。此 dcmd 要求在 kmem_flags 中设置 KMF_AUDIT 标志。

  如果指定了一个或多个高速缓存名称（例如 kmem_alloc_256），则对内存使用情况的扫描仅限于那些高速缓存。 缺省情况下，包括所有高速缓存。 如果使用了 -e 选项，则包括分配器的小用户。 小用户是总数小于 1024 字节内存的分配或者同一栈跟踪中的分配数小于 10 的分配。 如果使用了 -f 选项，则会针对每个单独的分配列显栈跟踪。

[*address*] ::kmem_cache
: 对指定地址上存储的 kmem_cache 结构或完整的活动 kmem_cache 结构集进行格式设置和显示。

::kmem_log
: 显示完整的 kmem 事务日志集（按反向时间顺序排列）。 此 dcmd 使用比 ::kmalog 更简明的表格输出格式。

[*address*] ::kmem_verify
验证在指定地址上存储的 kmem_cache 结构或完整的活动 kmem_cache 结构集的完整性。如果指定了显式高速缓存地址，则 dcmd 会显示有关错误的详细信息；否则，显示摘要报告。第 83 页中的 "内核内存高速缓存"中对 ::kmem_verify dcmd 进行了更详细的讨论。

[*address*] ::vmem
对在指定地址上存储的 vmem 结构或完整的活动 vmem 结构集进行格式设置和显示。此结构在 <sys/vmem_impl.h> 中定义。

address ::vmem_seg
对在指定地址上存储的 vmem_seg 结构进行格式设置和显示。此结构在 <sys/vmem_impl.h> 中定义。

address ::whatis [-abv]
报告有关指定地址的信息。 具体来说，::whatis 会尝试确定该地址是指向 kmem 管理的缓冲区的指针还是其他类型的特殊内存区域（如线程栈），然后报告其判定结果。 如果存在 -a 选项，则 dcmd 会向其查询报告所有匹配项，而不仅仅是第一个匹配项。 如果存在 -b 选项，则 dcmd 还会尝试确定该地址是否通过已知的 kmem bufctl 被引用。 如果存在 -v 选项，则 dcmd 在搜索各种内核数据结构时会报告其进度。

Walker

allocdby
在指定 kthread_t 结构的地址作为起点的情况下，迭代与此内核线程执行的内存分配相对应的 bufctl 结构集。

bufctl
在指定 kmem_cache_t 结构的地址作为起点的情况下，迭代与此高速缓存关联的分配的 bufctl 集。

freectl
在指定 kmem_cache_t 结构的地址作为起点的情况下，迭代与此高速缓存关联的空闲 bufctl 集。

freedby
在指定 kthread_t 结构的地址作为起点的情况下，迭代与此内核线程执行的内存取消分配相对应的 bufctl 结构集。

freemem
在指定 kmem_cache_t 结构的地址作为起点的情况下，迭代与此高速缓存关联的空闲缓冲区集。

kmem
在指定 kmem_cache_t 结构的地址作为起点的情况下，迭代与此高速缓存关联的分配的缓冲区集。

kmem_cache
迭代活动的 kmem_cache_t 结构集。 此结构在 <sys/kmem_impl.h> 中定义。

kmem_cpu_cache
在指定 kmem_cache_t 结构的地址作为起点的情况下，迭代与此高速缓存关联的每个 CPU 的 kmem_cpu_cache_t 结构。 此结构在 <sys/kmem_impl.h> 中定义。

kmem_slab
在指定 kmem_cache_t 结构的地址作为起点的情况下，迭代关联的 kmem_slab_t 结构集。 此结构在 <sys/kmem_impl.h> 中定义。

kmem_log
迭代存储在 kmem 分配器事务日志中的 bufctl 集。

leak	在指定 bufctl 结构的地址的情况下，迭代与具有类似分配栈跟踪的泄漏内存缓冲区相对应的 bufctl 结构集。必须首先应用 ::findleaks dcmd 查找内存泄漏，然后才能使用 leak walker。
leakbuf	在指定 bufctl 结构的地址的情况下，迭代与具有类似分配栈跟踪的泄漏内存缓冲区相对应的缓冲区地址集。必须首先应用 ::findleaks dcmd 查找内存泄漏，然后才能使用 leakbuf walker。

文件系统

MDB 文件系统调试支持包括一个内置工具，用于将 vnode 指针转换为对应文件系统的路径名。 此转换是使用目录名称查找高速缓存 (Directory Name Lookup Cache, DNLC) 执行的；由于高速缓存并不包含所有的活动 vnode， 因此可能无法将某些 vnode 转换为路径名， 并且显示的是 "??" 而不是名称。

dcmd

::fsinfo	显示挂载的文件系统表， 包括 vfs_t 地址、ops 向量和每个文件系统的挂载点。
::lminfo	显示已向锁定管理器注册其活动网络锁定的 vnode 表。 将会显示对应于每个 vnode 的路径名。
address ::vnode2path [-v]	显示对应于指定 vnode 地址的路径名。如果指定了 -v 选项，则 dcmd 列显更详细的显示，其中包括每个中间路径组件的 vnode 指针。

Walker

buf	迭代活动的块 I/O 传输结构集（buf_t 结构）。 buf 结构在 <sys/buf.h> 中定义， buf(9S) 中对其进行了更详细的介绍。

虚拟内存

本节介绍对内核虚拟内存子系统的调试支持。

dcmd

address ::addr2smap [*offset*]	列显与内核 segmap 地址空间段中的指定地址相对应的 smap 结构地址。
as ::as2proc	显示与 as_t 地址 *as* 相对应的进程的 proc_t 地址。
[*address*] ::memlist [-aiv]	显示指定的 memlist 结构或已知的 memlist 结构之一。 如果不存在 memlist 地址和选项， 或者如果存在 -i 选项， 则显示

	表示以物理方式安装的内存的 `memlist`。 如果存在 `-a` 选项，则显示表示可用物理内存的 `memlist`。 如果存在 `-v` 选项，则显示表示可用虚拟内存的 memlist。
`::memstat`	显示系统范围的内存使用情况摘要。 将会显示不同类别的页面（内核、匿名内存、可执行文件和库、页面高速缓存和空闲列表）所占用的系统内存量和百分比，以及总系统内存量。
[*address*] `::page`	显示指定的 `page_t` 的属性。 如果未指定任何 `page_t` 地址，则 dcmd 会显示所有系统页面的属性。
seg `::seg`	对指定的地址空间段（`seg_t` 地址）进行格式设置和显示。
[*address*] `::swapinfo`	显示有关所有活动的 `swapinfo` 结构或指定结构 `swapinfo` 的信息。 将会显示每个结构的 vnode、文件名和统计信息。
vnode `::vnode2smap` [*offset*]	列显与指定的 `vnode_t` 地址和偏移相对应的 `smap` 结构地址。

Walker

anon	在指定 `anon_map` 结构的地址作为起点的情况下，迭代相关的 anon 结构集。 anon 映射实现在 `<vm/anon.h>` 中定义。
memlist	迭代指定的 `memlist` 结构的各个跨度。 此 walker 可以与 `::memlist` dcmd 联合使用以显示每个跨度。
page	迭代所有系统的 `page` 结构。 如果为 walk 指定了显式地址，则会将其视为 vnode 的地址，并且 walker 仅迭代与该 vnode 关联的那些页面。
seg	在指定 `as_t` 结构的地址作为起点的情况下，迭代与指定地址空间关联的地址空间段集（`seg` 结构）。 `seg` 结构在 `<vm/seg.h>` 中定义。
swapinfo	迭代活动 `swapinfo` 结构的列表。此 walker 可以与 `::swapinfo` dcmd 联合使用。

CPU 和分发程序

本节介绍用于检查 cpu 结构和内核分发程序的状态的工具。

dcmd

`::callout`	显示调用表。将会显示每次调用的函数、参数和失效时间。
`::class`	显示调度类表。
[*cpuid*] `::cpuinfo` [`-v`]	显示当前在每个 CPU 上执行的线程表。 如果在 dcmd 名称前面指定了可选的 CPU ID 号或 CPU 结构地址，则仅显示有关指定 CPU 的信息。如果存在 `-v` 选项，则 `::cpuinfo` 还会显示等待在每个 CPU 上执行的可运行线程以及活动的中断线程。

Walker

cpu　迭代内核 CPU 结构集。`cpu_t` 结构在 `<sys/cpuvar.h>` 中定义。

设备驱动程序和 DDI 框架

本节介绍对内核开发者以及第三方设备驱动程序开发者均有所帮助的 dcmd 和 walker。

dcmd

address `::binding_hash_entry`
在指定内核名称到主设备号的绑定散列表项的地址（结构 `bind`）的情况下，显示节点绑定名称、主设备号和指向下一个元素的指针。

`::devbindings` *device-name*
显示指定驱动程序的所有实例的列表。输出包括对应于每个实例的项（从指向结构 `dev_info` 的指针开始，可使用 `$<devinfo` 或 `::devinfo` 查看此结构）、驱动程序名称、实例编号以及与该实例关联的驱动程序和系统属性。

address `::devinfo` [`-q`]
列显与 devinfo 节点关联的系统和驱动程序属性。如果指定了 `-q` 选项，则仅显示设备节点的核心摘要。

address `::devinfo2driver`
列显与 `devinfo` 节点关联的驱动程序的名称（如果有）。

[*address*] `::devnames` [`-v`]
显示内核的 `devnames` 表以及指向驱动程序实例列表的 `dn_head` 指针。 如果指定了 `-v` 标志，则显示在 `devnames` 表的各项中存储的其他信息。

[*devinfo*] `::prtconf` [`-cpv`]
从 *devinfo* 指定的设备节点开始显示内核设备树。 如果未提供 *devinfo*，则缺省情况下假定从设备树的根开始。 如果指定了 `-c` 选项，则仅显示指定设备节点的子节点。 如果指定了 `-p` 选项，则仅显示指定设备节点的祖先。 如果指定了 `-v`，则显示与每个节点关联的属性。

[*major-num*] `::major2name` [*major-num*]
显示与指定的主设备号相对应的驱动程序名称。可以将主设备号指定为 dcmd 前面的表达式或指定为命令行参数。

[*address*] `::modctl2devinfo`
列显与指定的 `modctl` 地址相对应的所有设备节点。

`::name2major` *driver-name*
在指定设备驱动程序名称的情况下，显示其主设备号。

[*address*] `::softstate` [*instance-number*]
在指定 softstate 状态指针（请参见 `ddi_soft_state_init`(9F)）和设备实例编号的情况下，显示该实例的软状态。

Walker

binding_hash
: 在指定内核绑定散列表项的数组的地址（结构 `bind **`）的情况下，遍历散列表中的所有项，并返回每个结构 `bind` 的地址。

devinfo
: 首先，迭代指定 devinfo 的父级，然后按级别从高到低的顺序将其返回。其次，返回指定的 devinfo 本身。第三，按级别从高到低的顺序迭代指定 devinfo 的子级。 `dev_info` 结构在 `<sys/ddi_impldefs.h>` 中定义。

devinfo_children
: 首先，返回指定的 devinfo，然后按级别从高到低的顺序迭代指定 devinfo 的子级。`dev_info` 结构在 `<sys/ddi_impldefs.h>` 中定义。

devinfo_parents
: 按级别从高到低的顺序迭代指定 devinfo 的父级，然后返回指定的 devinfo。 `dev_info` 结构在 `<sys/ddi_impldefs.h>` 中定义。

devi_next
: 迭代指定 devinfo 的同级。`dev_info` 结构在 `<sys/ddi_impldefs.h>` 中定义。

devnames
: 迭代 devnames 数组中的项。此结构在 `<sys/autoconf.h>` 中定义。

softstate
: 在指定 softstate 指针（请参见 `ddi_soft_state_init` (9F)）的情况下，显示指向驱动程序状态结构的所有非 NULL 指针。

softstate_all
: 在指定 softstate 指针（请参见 `ddi_soft_state_init`(9F)）的情况下，显示指向驱动程序状态结构的所有指针。请注意，未使用的实例的指针将为 NULL。

STREAMS

本节介绍对内核开发者以及第三方 STREAMS 模块和驱动程序的开发者均有所帮助的 dcmd 和 walker。

dcmd

address `::mblk2dblk`
: 在指定 `mblk_t` 的地址的情况下，列显对应的 `dblk_t` 的地址。

[*address*] `::mblk_verify`
: 验证一个或多个消息块的完整性。如果指定了显式消息块地址，则检查此消息块的完整性。如果未指定任何地址，则检查所有活动消息块的完整性。 此 dcmd 可生成所检测到的任何无效消息块状态的输出。

address `::queue` [`-v`] [`-f` *flag*] [`-F` *flag*] [`-s` *syncq*]
: 过滤并显示指定的 `queue_t` 数据结构。 如果没有任何选项，则显示 `queue_t` 的各种属性。 如果存在 `-v` 选项，则更详细地解码队列标志。 如果存在 `-f`、`-F` 或 `-m` 选项，则仅当队列与这些选项的参数所定义的条件匹配时才显示该队列；通过此方式，dcmd 可用作管道输入的过滤器。 `-f` 选项表示指定的标志（`<sys/stream.h>` 中的其中一个 Q 标志名称）

必须存在于队列标志中。 -F 选项表示队列标志中决不能存在指定的标志。-m 选项表示与队列关联的模块名称必须与指定的 modname 匹配。-s 选项表示与队列关联的 syncq_t 必须与指定的 syncq_t 地址匹配。

address ::q2syncq
在指定 queue_t 的地址的情况下，列显对应的 syncq_t 数据结构的地址。

address ::q2otherq
在指定 queue_t 的地址的情况下，列显对等的读取或写入队列结构的地址。

address ::q2rdq
在指定 queue_t 的地址的情况下，列显对应的读取队列的地址。

address ::q2wrq
在指定 queue_t 的地址的情况下，列显对应的写入队列的地址。

[*address*] ::stream
在 stdata_t 结构的地址表示 STREAM 头的情况下，显示内核 STREAM 数据结构的图像。 将会显示每个模块的读取和写入队列指针、字节计数以及标志，在某些情况下，还会在页边距中显示特定队列的其他信息。

address ::syncq [-v] [-f *flag*] [-F *flag*] [-t *type*] [-T *type*]
过滤并显示指定的 syncq_t 数据结构。 如果没有任何选项，则显示 syncq_t 的各种属性。 如果存在 -v 选项，则更详细地解码 syncq 标志。 如果存在 -f、-F、-t 或 -T 选项，则仅当 syncq 与这些选项的参数所定义的条件匹配时才显示它；通过这种方式，dcmd 可用作管道输入的过滤器。-f 选项表示指定的标志（<sys/strsubr.h> 中的其中一个 SQ_ 标志名称）必须存在于 syncq 标志中。-F 选项表示 syncq 标志中决不能存在指定的标志。-t 选项表示指定的类型（<sys/strsubr.h> 中的其中一个 SQ_CI 或 SQ_CO 类型名称）必须存在于 syncq 类型位中。 -T 选项表示 syncq 类型位中决不能存在指定的类型。

address ::syncq2q
在指定 syncq_t 的地址的情况下，列显对应的 queue_t 数据结构的地址。

Walker

b_cont　在指定 mblk_t 的地址的情况下，通过跟踪 b_cont 指针迭代关联的消息结构集。b_cont 指针用于将指定的消息块链接到为同一消息的延续的下一个关联消息块。msgb(9S) 中对消息块进行了更详细的介绍

b_next　在指定 mblk_t 的地址的情况下，通过跟踪 b_next 指针迭代关联的消息结构集。b_next 指针用于将指定的消息块链接到指定队列中的下一个关联消息块。msgb(9S) 中对消息块进行了更详细的介绍。

qlink　在指定 queue_t 结构的地址的情况下，使用 q_link 指针遍历相关队列的列表。此结构在 <sys/stream.h> 中定义。

qnext　在指定 queue_t 结构的地址的情况下，使用 q_next 指针遍历相关队列的列表。此结构在 <sys/stream.h> 中定义。

readq　在指定 stdata_t 结构的地址的情况下，遍历读面队列结构的列表。

writeq　在指定 stdata_t 结构的地址的情况下，遍历写面队列结构的列表。

联网

以下提供的 dcmd 和 walker 有助于调试核心内核联网栈协议。

dcmd

address `::mi` [`-p`] [`-d` | `-m`]
: 在指定内核 MI_O 的地址的情况下，过滤并显示 MI_O 或其有效负荷。如果指定了 `-p` 选项，则显示 MI_O 对应的有效负荷的地址，否则显示 MI_O 本身。通过指定过滤器 `-d` 或 `-m`，dcmd 可以过滤设备或模块 MI_O 对象。

`::netstat` [`-av`] [`-f` `inet` | `inet6` | `unix`] [`-P` `tcp` | `udp`]
: 显示网络统计信息和活动连接。 如果存在 `-a` 选项，则显示所有套接字的状态。 如果存在 `-v` 选项，则显示更详细的输出。 如果存在 `-f` 选项，则仅显示与指定的地址族关联的连接。如果存在 `-P` 选项，则仅显示与指定协议关联的连接。

[*address*] `::sonode` [`-f` `inet` | `inet6` | `unix` | *id*] [`-t` `stream` | `dgram` | `raw` | *id*] [`-p` *id*]
: 过滤并显示 sonode 对象。如果未指定任何地址，则显示 AF_UNIX 套接字的列表，否则仅显示指定的 sonode。 如果存在 `-f` 选项，则仅输出指定族的套接字。如果存在 `-t` 选项，则仅输出指定类型的 sonode。如果存在 `-p` 选项，则仅显示指定协议的套接字。

[*address*] `::tcpb` [`-av`] [`-P` `v4` | `v6`]
: 过滤并显示 tcpb 对象。如果未指定任何地址，则遍历所有连接，否则仅过滤/显示指定的 tcpb。如果仅为活动连接指定 `-a` 过滤器，则可以使用 `-P` 过滤 TCP IPv4 或 IPv6 连接。`tcpb` dcmd 可以对 TCP 连接进行智能过滤，如果 IPv6 TCP 连接处于仍然允许 IPv4 连接的状态，则 `-P` 过滤器会将该连接同时视为 IPv4 和 IPv6，这与 `::netstat` 所使用的方式非常相似。如果未将 dcmd 用作过滤器并且指定了 `-v` 选项，则 dcmd 的输出将为详细输出。

Walker

ar
: 此 walker 用于在指定 ar 的地址的情况下，遍历从指定 ar 到最后一个 ar 的所有 ar 对象。如果未指定任何地址，则遍历所有 ar 对象。

icmp
: 此 walker 用于在指定 icmp 的地址的情况下，遍历从指定 icmp 到最后一个 icmp 的所有 icmp 对象。如果未指定任何地址，则遍历所有 icmp 对象。

ill
: 此 walker 用于在指定接口链路层 (interface link layer, ill) 结构的地址的情况下，遍历从指定 ill 到最后一个 ill 的所有 ill 对象。如果未指定任何地址，则遍历所有 ill 对象。

ipc
: 此 walker 用于在指定 ipc 的地址的情况下，遍历从指定 ipc 到最后一个 ipc 的所有 ipc 对象。如果未指定任何地址，则遍历所有 ipc 对象。

mi
: 在指定 MI_O 的地址情况下，遍历此 MI 中的所有 MI_O。

sonode
: 在指定 AF_UNIX sonode 的地址的情况下，从指定的 sonode 开始遍历 AF_UNIX sonode 的关联列表。如果未指定任何地址，则此 walker 将遍历所有 AF_UNIX 套接字的列表。

tcpb 此 walker 用于在指定 tcpb 的地址的情况下，遍历从指定 tcpb 到最后一个 TCP 连接的所有 TCP 连接。如果未指定任何地址，则遍历所有 tcpb 对象。

udp 此 walker 用于在指定 udp 的地址的情况下，遍历从指定 udp 到最后一个 udp 的所有 udp 对象。如果未指定任何地址，则遍历所有 udp 对象。

文件、进程和线程

本节介绍用于对 Solaris 内核中各种基础文件、进程和线程结构进行格式设置和检查的 dcmd 和 walker。

dcmd

process `::fd` *fd-num*
列显对应于与指定进程关联的文件描述符 *fd-num* 的 `file_t` 地址。 进程是使用其 `proc_t` 结构的虚拟地址指定的。

thread `::findstack` [*command*]
列显与指定内核线程（通过其 `kthread_t` 结构的虚拟地址标识）关联的栈跟踪。 该 dcmd 使用几种不同的算法查找相应的栈反向跟踪。 如果指定了可选的命令字符串，则将点变量重置为最顶部栈帧的帧指针地址，并且计算指定的命令，就好像已在命令行中键入它一样。 缺省的命令字符串是 `"<.$C0"`；即用于列显包括帧指针（但不包括参数）的栈跟踪。

`::pgrep` [`-no`] *regexp*
显示其名称与 *regexp* 正则表达式模式匹配的进程的进程信息。如果存在 `-n` 选项，则仅显示与模式匹配的最新进程。如果存在 `-o` 选项，则仅显示与模式匹配的最旧进程。

pid `::pid2proc`
列显与指定的 PID 相对应的 `proc_t` 地址。 请记住 MDB 的缺省基数是十六进制的，因此使用 `pgrep`(1) 或 `ps`(1) 获取的十进制 PID 应该使用 `0t` 作为前缀。

process `::pmap` [`-q`]
列显指定的进程地址所指示的进程的内存映射。该 dcmd 使用与 `pmap`(1) 类似的格式显示输出。如果存在 `-q` 选项，则 dcmd 显示其输出的缩写形式（所需的处理时间较短）。

[*address*] `::ps` [`-fltTP`]
列显与指定的进程或所有活动系统进程相关的信息摘要，与 `ps`(1) 类似。如果指定了 `-f` 选项，则列显完整的命令名和初始参数。如果指定了 `-l` 选项，则列显与每个进程关联的 LWP。 如果指定了 `-t` 选项，则列显与每个进程 LWP 关联的内核线程。如果指定了 `-T` 选项，则显示与每个进程关联的任务 ID。如果指定了 `-P` 选项，则显示与每个进程关联的项目 ID。

`::ptree`
列显进程树，其中子进程从其各自的父进程中缩进。 该 dcmd 使用与 `ptree`(1) 类似的格式显示输出。

address ::task
列显活动内核任务结构及其关联 ID 号和属性的列表。settaskid(2) 中对进程任务 ID 进行了更详细的介绍。

[*address*] ::thread [-bdfimps]
显示指定内核 kthread_t 结构的属性。 如果未指定 kthread_t 地址，则显示所有内核线程的属性。 dcmd 选项用于控制所显示的输出列。 如果不存在任何选项，则缺省情况下会启用 -i 选项。 如果存在 -b 选项，则显示与线程的十字转门 (turnstile) 和阻塞同步对象相关的信息。如果存在 -d 选项，则显示线程的分发程序优先级、绑定和上次分发时间。 如果存在 -f 选项，则输出中不显示状态为 TS_FREE 的线程。 如果存在 -i 选项（缺省选项），则显示线程状态、标志、优先级和中断信息。 如果存在 -m 选项，则所有其他输出选项都将合并到一行输出上。 如果存在 -p 选项，则显示线程的进程、LWP 和凭证指针。 如果存在 -s 选项，则显示线程的信号队列以及暂挂信号和保留信号的掩码。

vnode ::whereopen
在指定 vnode_t 地址的情况下，列显当前在其文件表中打开了此 vnode 的所有进程的 proc_t 地址。

Walker

file 在将 proc_t 结构的地址作为起点的情况下，迭代与指定进程关联的打开文件集（file_t 结构）。 file_t 结构在 <sys/file.h> 中定义。

proc 迭代活动进程 (proc_t) 结构。 此结构在 <sys/proc.h> 中定义。

任务 在指定任务指针的情况下，迭代作为指定任务成员的进程的 proc_t 结构列表。

thread 迭代内核线程 (kthread_t) 结构集。 如果调用了全局遍历，则 walker 将返回所有内核线程。如果使用 proc_t 地址作为起点来调用局部遍历，则将返回与指定进程关联的线程集。kthread_t 结构在 <sys/thread.h> 中定义。

同步元语

本节介绍用于检查特定的内核同步元语的 dcmd 和 walker。 每个元语的语义在手册页对应的 (9f) 部分中讨论。

dcmd

rwlock ::rwlock 在指定读取器-写入器锁的地址（请参见 rwlock(9F)）的情况下，显示锁的当前状态和等待线程的列表。

address ::sobj2ts 将同步对象的地址转换为对应的十字转门 (turnstile) 地址，并列显该十字转门 (turnstile) 地址。

[*address*] ::turnstile 显示指定的 turnstile_t 的属性。如果未指定 turnstile_t 地址，则 dcmd 显示所有十字转门 (turnstile) 的属性。

[*address*] ::wchaninfo [-v]	在指定条件变量（请参见 condvar(9F)）或信号（请参见 semaphore(9F)）的地址的情况下，显示当前等待此对象的线程的数量。 如果未指定任何显式地址，则显示所有包含等待线程的对象。 如果指定了 -v 选项，则显示在每个对象上阻塞的线程的列表。

Walker

blocked	在指定同步对象（如 mutex(9F) 或 rwlock(9F)）的地址的情况下，迭代阻塞的内核线程的列表。

wchan	在指定条件变量（请参见 condvar(9F)）或信号（请参见 semaphore(9F)）的地址的情况下，迭代阻塞的内核线程的列表。

循环

循环子系统是一种底层内核子系统，用于为其他内核服务和编程接口提供高精度、按 CPU 间隔计时器功能。

dcmd

::cycinfo [-vV]	按每个 CPU 的 CPU 状态显示循环子系统。如果存在 -v 选项，则显示更显示的内容。如果存在 -V 选项，则显示比使用 -v 时更详细的内容。

address ::cyclic	对指定地址上的 cyclic_t 进行格式设置和显示。

::cyccover	显示循环子系统代码的适用范围信息。 此信息仅在 DEBUG 内核中可用。

::cyctrace	显示循环子系统的跟踪信息。 此信息仅在 DEBUG 内核中可用。

Walker

cyccpu	迭代每个 CPU 的 cyc_cpu_t 结构。 此结构在 <sys/cyclic_impl.h> 中定义。

cyctrace	迭代循环跟踪缓冲区结构。 此信息仅在 DEBUG 内核中可用。

任务队列

任务队列子系统可为内核中的各种客户机提供通用的异步任务调度。

dcmd

address ::taskq_entry	列显指定结构 taskq_entry 的内容。

Walker

taskq_entry　在指定 taskq 结构的地址的情况下，迭代 taskq_entry 结构的列表。

错误队列

错误队列子系统可为平台特定的错误处理代码提供通用的异步错误事件处理。

dcmd

[*address*] ::errorq　显示与指定的错误队列相关的信息摘要。如果未指定任何地址，则显示与所有系统错误队列相关的信息。 将会显示每个队列的地址、名称、队列长度和数据元素大小，以及各种队列统计信息。

Walker

errorq　遍历系统错误队列的列表，并返回每个单独错误队列的地址。

errorq_data　在指定错误队列的地址的情况下，返回每个暂挂错误事件数据缓冲区的地址。

配置

本节介绍可以用于检查系统配置数据的 dcmd。

dcmd

::system　显示内核在系统初始化过程中解析文件时 system(4) 配置文件的内容。

进程间通信调试支持 (ipc)

ipc 模块可为实现消息队列、信号和共享内存进程间通信元语提供调试支持。

dcmd

::ipcs [-l]　显示系统范围的 IPC 标识符的列表，这些标识符对应于已知的消息队列、信号和共享内存段。 如果指定了 -l 选项，则显示较长的信息列表。

address ::msg [-l] [-t *type*]　显示指定的消息队列元素（结构 msg）的属性。 如果存在 -l 选项，则使用十六进制和 ASCII 显示消息的原始内容。 如果

存在 -t 选项，则可以将其用于过滤输出并仅显示指定类型的消息。 这可能有助于将 msgqueue walker 的输出传输到 ::msg。

id ::msqid [-k] 将指定的消息队列 IPC 标识符转换为指向对应内核实现结构的指针，并列显此内核结构的地址。 如果存在 -k 选项，则将 *id* 相应地解释为要匹配的消息队列键（请参见 msgget(2)）。

[*address*] ::msqid_ds [-l] 列显指定的 msqid_ds 结构或活动 msqid_ds 结构（消息队列标识符）表。 如果指定了 -l 选项，则显示较长的信息列表。

id ::semid [-k] 将指定的信号 IPC 标识符转换为指向对应内核实现结构的指针，并列显此内核结构的地址。 如果存在 -k 选项，则将 *id* 相应地解释为要匹配的信号键（请参见 semget(2)）。

[*address*] ::semid_ds [-l] 列显指定的 semid_ds 结构或活动 semid_ds 结构（信号标识符）表。 如果指定了 -l 选项，则显示较长的信息列表。

id ::shmid [-k] 将指定的共享内存 IPC 标识符转换为指向对应内核实现结构的指针，并列显此内核结构的地址。 如果存在 -k 选项，则将 *id* 相应地解释为要匹配的共享内存键（请参见 shmget(2)）。

[*address*] ::shmid_ds [-l] 列显指定的 shmid_ds 结构或活动 shmid_ds 结构（共享内存段标识符）表。 如果指定了 -l 选项，则显示较长的信息列表。

Walker

msg 遍历与消息队列标识符相对应的活动 msqid_ds 结构。此结构在 <sys/msg.h> 中定义。

msgqueue 迭代当前在指定消息队列中排队的 message 结构。

sem 遍历与信号标识符相对应的活动 semid_ds 结构。此结构在 <sys/sem.h> 中定义。

shm 遍历与共享内存段标识符相对应的活动 shmid_ds 结构。此结构在 <sys/shm.h> 中定义。

回送文件系统调试支持 (lofs)

lofs 模块可为 lofs(7FS) 文件系统提供调试支持。

dcmd

[*address*] ::lnode 列显指定的 lnode_t 或内核中活动 lnode_t 结构的表。

address ::lnode2dev　　列显与指定的 lnode_t 地址相对应的基础回送挂载文件系统的 dev_t (vfs_dev)。

address ::lnode2rdev　　列显与指定的 lnode_t 地址相对应的基础回送挂载文件系统的 dev_t (li_rdev)。

Walker

lnode　　遍历内核中的活动 lnode_t 结构。 此结构在 <sys/fs/lofs_node.h> 中定义。

Internet 协议模块调试支持 (ip)

ip 模块可为 ip(7P) 驱动程序提供调试支持

dcmd

[*address*] ::ire [-q]　　列显指定的 ire_t 或内核中活动 ire_t 结构的表。 如果指定了 -q 标志，则列显发送和接收队列指针而不是源地址和目标地址。

Walker

ire　　遍历内核中的活动 ire（Internet Route Entry，Internet 路由项）结构。此结构在 <inet/ip.h> 中定义。

内核运行时链接编辑器调试支持 (krtld)

本节介绍对内核运行时链接编辑器的调试支持，此编辑器负责装入内核模块和驱动程序。

dcmd

[*address*] ::modctl　　列显指定的 modctl 或内核中活动 modctl 结构的表。

address ::modhdrs　　在指定 modctl 结构的地址的情况下，列显模块的 ELF 可执行文件头和节头。

::modinfo　　列显有关活动内核模块的信息，类似于 /usr/sbin/modinfo 命令的输出。

Walker

modctl　遍历内核中活动 modctl 结构的列表。 此结构在 <sys/modctl.h> 中定义。

USB 框架调试支持 (uhci)

uchi 模块可为通用串行总线 (Universal Serial Bus, USB) 框架的主机控制器接口部分提供调试支持。

dcmd

address ::uhci_qh [-bd]　在指定 USB UHCI 控制器队列头 (Queue Head, QH) 结构的地址的情况下，列显此结构的内容。如果存在 -b 选项，则迭代 link_ptr 链，从而列显找到的所有 QH。 如果存在 -d 选项，则迭代 element_ptr 链，从而列显找到的所有 TD。

address ::uhci_td [-d]　在指定 USB UHCI 控制器事务描述符 (Transaction Descriptor, TD) 结构的地址的情况下，列显此结构的内容。 请注意，这仅适用于控制 TD 和中断 TD。 如果存在 -d 选项，则迭代 element_ptr 链，从而列显找到的所有 TD。

Walker

uhci_qh　在指定 USB UHCI 控制器队列头 (Queue Head, QH) 结构的地址的情况下，迭代此类结构的列表。

uhci_td　在指定 USB UHCI 控制器队列头描述符 (TD) 结构的地址的情况下，迭代此类结构的列表。

USB 框架调试支持 (usba)

usba 模块可为与平台无关的通用串行总线 (Universal Serial Bus, USB) 框架提供调试支持。

dcmd

::usba_debug_buf　列显 USB 调试信息缓冲区。

address ::usb_hcdi_cb　在指定主机控制器回调结构的地址（结构 usb_hcdi_cb）的情况下，列显此回调的摘要信息。

[*address*] ::usb_device [-pv]	在指定 usb_device 结构的地址的情况下，列显摘要信息。如果未提供任何地址，则此 dcmd 将遍历 usb_device 结构的全局列表。 如果存在 -p 选项，则还将列出此设备上所有打开的管道的信息。 如果存在 -v 选项，则列出每个设备的详细信息。
address ::usb_pipe_handle	在指定 USB 管道句柄结构（结构 usb_pipe_handle_impl）的地址的情况下，列显此句柄的摘要信息。

Walker

usb_hcdi_cb	在指定 USB 主机控制器 devinfo 节点的地址的情况下，迭代控制器的 usb_hcdi_cb_t 结构列表。
usba_list_entry	在指定 usba_list_entry 结构的地址的情况下，迭代此类结构的链。

x86: x86 平台调试支持 (unix)

这些 dcmd 和 walker 专用于 x86 平台。

dcmd

[*cpuid* | *address*] ::ttrace [-x]

按反向时间顺序显示陷阱跟踪记录。 陷阱跟踪工具仅在 DEBUG 内核中可用。如果指定了显式的点值，则将其解释为 CPU ID 号或陷阱跟踪记录地址，具体取决于确切的值。如果指定了 CPU ID，则输出仅限于该 CPU 中的缓冲区。如果指定了记录地址，则仅设置该记录的格式。如果指定了 -x 选项，则显示完整的原始记录。

Walker

ttrace	按反向时间顺序遍历陷阱跟踪记录地址的列表。 陷阱跟踪工具仅在 DEBUG 内核中可用。

SPARC: sun4u 平台调试支持 (unix)

这些 dcmd 和 walker 专用于 SPARC sun4u 平台。

dcmd

[*address*] ::softint	显示指定地址上的软中断向量结构，或显示所有的活动软中断向量。 将会显示每个结构的暂挂计数、PIL、参数和处理程序函数。
::ttctl	显示陷阱跟踪控制记录。 陷阱跟踪工具仅在 DEBUG 内核中可用。
[*cpuid*] ::ttrace [-x]	按反向时间顺序显示陷阱跟踪记录。陷阱跟踪工具仅在 DEBUG 内核中可用。 如果指定了显式点值，则将其解释为 CPU ID 号，并且输出仅限于该 CPU 中的缓冲区。 如果指定了 -x 选项，则显示完整的原始记录。
[*address*] ::xc_mbox	显示指定地址上的交叉调用信箱，或设置具有暂挂请求的所有交叉调用信箱的格式。
::xctrace	按照与 CPU 交叉调用活动相关的反向时间顺序，对交叉调用跟踪记录进行格式设置和显示。交叉调用跟踪工具仅在 DEBUG 内核中可用。

Walker

softint	迭代软中断向量表的各项。
ttrace	按反向时间顺序迭代陷阱跟踪记录地址。 陷阱跟踪工具仅在 DEBUG 内核中可用。
xc_mbox	迭代用于 CPU 握手和交叉调用 (x-call) 请求的信箱。



第 9 章

使用内核内存分配器进行调试

Solaris 内核内存 (kmem) 分配器提供了一组强大的调试功能，可以简化内核崩溃转储的分析。本章将讨论这些调试功能以及专门为分配器设计的 MDB dcmd 和 walker。Bonwick（请参见第 10 页中的 “相关书籍和文章”）概述了分配器本身的原理。有关分配器数据结构的定义，请参阅头文件 `<sys/kmem_impl.h>`。可以在产品化的系统中启用 kmem 调试功能以增强问题分析，或者在开发系统中启用它以协助调试内核软件和设备驱动程序。

注 – 本指南反映了 Solaris 10 实现；此信息对于过去或将来的发行版可能不相关、不正确或不适用，因为它反映的是当前的内核实现。它未定义任何类型的公共接口。所提供的有关内核内存分配器的所有信息在将来的 Solaris 发行版中可能会更改。

入门：创建崩溃转储样例

本节说明如何获取崩溃转储样例以及如何调用 MDB 对其进行检查。

设置 kmem_flags

内核内存分配器包含许多高级调试功能，但是由于这些功能可能会导致性能下降，因此缺省情况下并未启用。为了理解本指南中的示例，您应该启用这些功能。应仅在测试系统中启用这些功能，因为它们可能会导致性能下降或暴露潜在的问题。

分配器的调试功能由 `kmem_flags` 可调参数控制。首先，请确保正确设置了 `kmem_flags`：

```
# mdb -k

> kmem_flags/X

kmem_flags:

kmem_flags:     f
```

如果未将 kmem_flags 设置为 'f'，则应该将以下行：

```
set kmem_flags=0xf
```

添加至 /etc/system，然后重新引导系统。系统重新引导时，请确认是否已将 kmem_flags 设置为 'f'。将此系统恢复为用于生产之前，请记住要删除对 /etc/system 的修改。

强制崩溃转储

下一步是确保正确配置了崩溃转储。首先，请确认是否将 dumpadm 配置为保存内核崩溃转储并启用了 savecore。 有关崩溃转储参数的更多信息，请参见 dumpadm(1M)。

```
# dumpadm

              Dump content: kernel pages

               Dump device: /dev/dsk/c0t0d0s1 (swap)

        Savecore directory: /var/crash/testsystem

          Savecore enabled: yes
```

接下来，使用 reboot(1M) 的 '-d' 标志重新引导系统，这将强制内核崩溃并保存崩溃转储。

```
# reboot -d

Sep 28 17:51:18 testsystem reboot: rebooted by root



panic[cpu0]/thread=70aacde0: forced crash dump initiated at user request



401fbb10 genunix:uadmin+55c (1, 1, 0, 6d700000, 5, 0)

  %l0-7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000

         00000000

...
```

系统重新引导时，请确保崩溃转储已成功：

```
$ cd /var/crash/testsystem

$ ls
```

```
bounds    unix.0    unix.1    vmcore.0  vmcore.1
```

如果转储目录中缺少该转储，可能是由于分区的空间不足。可以释放空间并以超级用户身份手动运行 savecore(1M)，以便随后保存转储。如果转储目录包含多个崩溃转储，则刚创建的转储将是具有最新修改时间的 unix.[n] 和 vmcore.[n] 对。

启动 MDB

现在，请对创建的崩溃转储运行 mdb 并检查其状态：

```
$ mdb unix.1 vmcore.1

Loading modules: [ unix krtld genunix ip nfs ipc ]

> ::status

debugging crash dump vmcore.1 (32-bit) from testsystem

operating system: 5.10 Generic (sun4u)

panic message: forced crash dump initiated at user request
```

在本指南提供的示例中，使用的是来自 32 位内核的崩溃转储。此处提供的所有方法都适用于 64 位内核，并且已仔细区分了指针（其大小在 32 位和 64 位系统中不同）与固定大小的数量（相对于内核数据模型不变）。

UltraSPARC 工作站用于生成所提供的示例。您得到的结果可能随所使用系统的体系结构和型号的不同而不同。

分配器基础知识

内核内存分配器的作用是将虚拟内存中的区域分配给其他内核子系统（它们通常称为**客户机**）。本节说明了分配器操作的基础知识，并介绍了本指南中稍后使用的一些术语。

缓冲区状态

内核内存分配器的作用域是虚拟内存中组成内核堆的**缓冲区**集。这些缓冲区将会分组为具有相同大小和用途的各缓冲区集（称为**高速缓存**）。每个高速缓存都包含一组缓冲区。其中的一些缓冲区当前**空闲**，这意味着尚未将其分配给分配器的任何客户机。其余的缓冲区**已分配**，这意味着已将指向该缓冲区的指针提供给分配器的客户机。如果分配器的所有客户机中都没有指向已分配的缓冲区的指针，则认为此缓冲区发生了**泄漏**，因为无法将其释放。泄漏的缓冲区表明错误的代码正在浪费内核资源。

事务

kmem **事务**是指缓冲区在已分配状态和空闲状态之间的转换。 分配器可以在每个事务执行过程中验证缓冲区的状态是否有效。此外，分配器还包含用于记录事务的工具，以便进行事后检查。

休眠分配和非休眠分配

与标准 C 库的 malloc(3C) 函数不同，内核内存分配器可以阻塞（或**休眠**），一直等到有足够的虚拟内存可用于满足客户机的请求为止。 这一行为由 kmem_alloc(9F) 的 'flag' 参数控制。调用设置了 KM_SLEEP 标志的 kmem_alloc(9F) 决不会失败；它将永远阻塞，等待资源变为可用。

内核内存高速缓存

内核内存分配器会将其管理的内存分为一组**高速缓存**。 所有分配都是从这些高速缓存（通过 kmem_cache_t 数据结构表示）实现的。每个高速缓存都具有固定的**缓冲区大小**，表示该高速缓存可提供的最大分配空间。 每个高速缓存都具有一个指明其管理的数据类型的字符串名称。

一些内核内存高速缓存具有特殊用途，并会进行初始化以便仅分配特定种类的数据结构。"thread_cache" 即是此类高速缓存的一个示例，它仅分配 kthread_t 类型的结构。 这些高速缓存中的内存通过 kmem_cache_alloc() 函数分配给客户机，并且通过 kmem_cache_free() 函数释放。

注 – kmem_cache_alloc() 和 kmem_cache_free() 不是公共的 DDI 接口。请勿编写依赖于这些接口的代码，因为将来的 Solaris 发行版中可能会更改或删除这些接口。

名称以 "kmem_alloc_" 开头的高速缓存可实现内核的常规内存分配方案。 这些高速缓存为 kmem_alloc(9F) 和 kmem_zalloc(9F) 的客户机提供内存。 其中的每个高速缓存都满足大小介于此类高速缓存的缓冲区大小和第二小的高速缓存的缓冲区大小之间的请求。 例如，内核具有 kmem_alloc_8 和 kmem_alloc_16 高速缓存。 在这种情况下，kmem_alloc_16 高速缓存可处理大小为 9-16 个字节内存的所有客户端请求。 请记住，无论客户端请求的大小是多少，kmem_alloc_16 高速缓存中每个缓冲区的大小均为 16 个字节。对于大小为 14 个字节的请求，所得到缓冲区中有两个字节是未使用的，因为该请求是从 kmem_alloc_16 高速缓存得到满足的。

最后一组高速缓存是指由内核内存分配器在内部使用以对其自身进行记录的高速缓存。 这包括名称以 "kmem_magazine_" 或 "kmem_va_"、kmem_slab_cache、kmem_bufctl_cache 等开头的那些高速缓存。

内核内存高速缓存

本节说明如何查找和检查内核内存高速缓存。通过发出 ::kmastat 命令，可以了解系统中的各种 kmem 高速缓存。

```
> ::kmastat

cache                        buf    buf    buf    memory     alloc alloc

name                        size in use  total    in use   succeed  fail

------------------------- ------ ------ ------ --------- --------- -----

kmem_magazine_1                8     24   1020      8192        24     0

kmem_magazine_3               16    141    510      8192       141     0

kmem_magazine_7               32     96    255      8192        96     0

...

kmem_alloc_8                   8   3614   3751     90112   9834113     0

kmem_alloc_16                 16   2781   3072     98304   8278603     0

kmem_alloc_24                 24    517    612     24576    680537     0

kmem_alloc_32                 32    398    510     24576    903214     0

kmem_alloc_40                 40    482    584     32768    672089     0

...

thread_cache                 368    107    126     49152    669881     0

lwp_cache                    576    107    117     73728       182     0

turnstile_cache               36    149    292     16384    670506     0

cred_cache                    96      6     73      8192   2677787     0

...
```

如果运行 ::kmastat，则可以了解“正常”系统的信息。 这将有助于发现系统中正在泄漏内存的过大高速缓存。 根据所运行的系统和正在运行的进程数等因素，::kmastat 的结果将有所不同。

列出各种 kmem 高速缓存的另一种方法是使用 ::kmem_cache 命令：

```
> ::kmem_cache

ADDR     NAME                      FLAG  CFLAG  BUFSIZE  BUFTOTL

70036028 kmem_magazine_1           0020 0e0000        8     1020

700362a8 kmem_magazine_3           0020 0e0000       16      510

70036528 kmem_magazine_7           0020 0e0000       32      255

...

70039428 kmem_alloc_8              020f 000000        8     3751

700396a8 kmem_alloc_16             020f 000000       16     3072

70039928 kmem_alloc_24             020f 000000       24      612

70039ba8 kmem_alloc_32             020f 000000       32      510

7003a028 kmem_alloc_40             020f 000000       40      584

...
```

此命令非常有用，因为它可将高速缓存名称映射到地址，并为 FLAG 列中的每个高速缓存提供调试标志。 必须要了解的是，分配器对调试功能的选择是基于每个高速缓存从这组标志派生而来的。 这些是在创建高速缓存时与全局 kmem_flags 变量一起设置的。在系统运行的同时设置 kmem_flags 不会影响调试行为，但会影响随后创建的高速缓存（这种情况在引导后很少发生）。

接下来，请直接使用 MDB 的 kmem_cache walker 遍历 kmem 高速缓存的列表：

```
> ::walk kmem_cache

70036028

700362a8

70036528

700367a8

...
```

这将产生对应于内核中每个 kmem 高速缓存的指针的列表。 要了解有关特定高速缓存的信息，请应用 kmem_cache 宏：

```
> 0x70039928$<kmem_cache
```

```
0x70039928:     lock
0x70039928:     owner/waiters
                0
0x70039930:     flags           freelist        offset
                20f             707c86a0        24
0x7003993c:     global_alloc    global_free     alloc_fail
                523             0               0
0x70039948:     hash_shift      hash_mask       hash_table
                5               1ff             70444858
0x70039954:     nullslab
0x70039954:     cache           base            next
                70039928        0               702d5de0
0x70039960:     prev            head            tail
                707c86a0        0               0
0x7003996c:     refcnt          chunks
                -1              0
0x70039974:     constructor     destructor      reclaim
                0               0               0
0x70039980:     private         arena           cflags
                0               104444f8        0
0x70039994:     bufsize         align           chunksize
                24              8               40
0x700399a0:     slabsize        color           maxcolor
                8192            24              32
```

```
0x700399ac:     slab_create     slab_destroy    buftotal

                3               0               612

0x700399b8:     bufmax          rescale         lookup_depth

                612             1               0

0x700399c4:     kstat           next            prev

                702c8608        70039ba8        700396a8

0x700399d0:     name    kmem_alloc_24

0x700399f0:     bufctl_cache    magazine_cache  magazine_size

                70037ba8        700367a8        15

...
```

用于调试的重要字段包括 'bufsize'、'flags' 和 'name'。kmem_cache 的名称（在本示例中为 "kmem_alloc_24"）指明了它在系统中的用途。 Bufsize 表示此高速缓存中每个缓冲区的大小；在本示例中，高速缓存用于进行大小为 24 或更小的分配。 'flags' 指明了为此高速缓存启用的调试功能。 可以找到在 <sys/kmem_impl.h> 中列出的调试标志。 在本示例中 'flags' 为 0x20f，即 KMF_AUDIT | KMF_DEADBEEF | KMF_REDZONE | KMF_CONTENTS | KMF_HASH。 本文档将在后续几节中说明每个调试功能。

如果有兴趣查看特定高速缓存中的缓冲区，可以直接遍历该高速缓存中已分配和已释放的缓存区：

```
> 0x70039928::walk kmem

704ba010

702ba008

704ba038

702ba030

...



> 0x70039928::walk freemem

70a9ae50
```

```
70a9ae28

704bb730

704bb2f8

...
```

MDB 提供了一种为 kmem walker 提供高速缓存地址的快捷方式：此方式会为每个 kmem 高速缓存提供特定的 walker，并且 walker 与高速缓存同名。例如：

```
> ::walk kmem_alloc_24

704ba010

702ba008

704ba038

702ba030

...



> ::walk thread_cache

70b38080

70aac060

705c4020

70aac1e0

...
```

现在，您已经知道如何迭代内核内存分配器的内部数据结构以及如何检查 `kmem_cache` 数据结构的最重要成员。

检测内存损坏

分配器的主要调试功能之一是它包括了用于快速识别数据损坏的算法。当检测到损坏时，分配器会导致系统立即出现故障。

本节介绍分配器如何识别数据损坏；您必须了解这一点才能调试这些问题。内存误用通常分为以下种类：

- 写入超出了缓冲区的结尾
- 访问未初始化的数据
- 继续使用已释放的缓冲区
- 损坏内核内存

阅读接下来的三节时，请牢记这些问题。 它们将有助于了解分配器的设计，并使您可以更有效地诊断问题。

检查已释放的缓冲区：0xdeadbeef

如果在 `kmem_cache` 的 flags 字段中设置了 `KMF_DEADBEEF (0x2)` 位，则分配器会尝试通过将特殊模式写入所有已释放缓冲区中，从而使内存损坏易于检测。 此模式即是 `0xdeadbeef`。 由于典型的内存区域同时包含已分配的内存和已释放的内存，因此每种块的各个节将是分散的；以下是来自 "`kmem_alloc_24`" 高速缓存的一个示例：

```
0x70a9add8:     deadbeef        deadbeef

0x70a9ade0:     deadbeef        deadbeef

0x70a9ade8:     deadbeef        deadbeef

0x70a9adf0:     feedface        feedface

0x70a9adf8:     70ae3260        8440c68e

0x70a9ae00:     5               4ef83

0x70a9ae08:     0               0

0x70a9ae10:     1               bbddcafe

0x70a9ae18:     feedface        139d

0x70a9ae20:     70ae3200        d1befaed

0x70a9ae28:     deadbeef        deadbeef

0x70a9ae30:     deadbeef        deadbeef
```

```
0x70a9ae38:     deadbeef        deadbeef

0x70a9ae40:     feedface        feedface

0x70a9ae48:     70ae31a0        8440c54e
```

从 `0x70a9add8` 开始的缓冲区是使用 `0xdeadbeef` 模式填充的，这就直接表明了缓冲区当前是空闲的。另一个空闲缓冲区从 `0x70a9ae28` 开始；一个已分配的缓冲区介于它们之间，位于 `0x70a9ae00`。

注 – 您可能已经注意到此内存区域布局中有一些空洞，3 个 24 字节区域应该仅占用 72 字节的内存，而不是此处显示的 120 字节。此差异将在下一节第 89 页中的 "Redzone（禁区）: `0xfeedface`"中进行说明。

Redzone（禁区）: `0xfeedface`

模式 `0xfeedface` 频繁地在以上缓冲区中出现。此模式称为 "redzone" 指示器。通过此模式，分配器（以及调试问题的程序员）可以确定"错误"代码是否超出了缓冲区的边界。redzone 后面是一些其他信息。该数据的内容取决于其他因素（请参见第 93 页中的 "内存分配日志记录"）。redzone 及其后缀统称为 *buftag* 区域。图 9–1 概述了此信息。

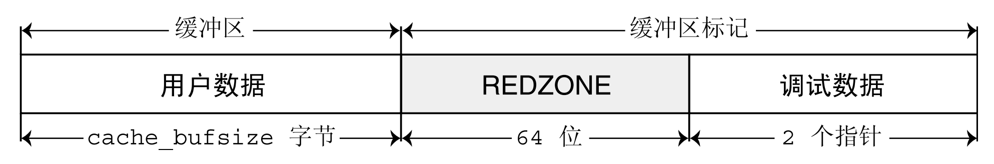


图 9–1 Redzone

如果在高速缓存中设置了 `KMF_AUDIT`、`KMF_DEADBEEF` 或 `KMF_REDZONE` 标志中的任何一个，则 buftag 会附加到该高速缓存的每个缓冲区。buftag 的内容取决于是否设置了 `KMF_AUDIT`。

现在，可以轻易将上述内存区域分解为不同的缓冲区：

```
0x70a9add8:     deadbeef        deadbeef   \

0x70a9ade0:     deadbeef        deadbeef    +- User Data (free)

0x70a9ade8:     deadbeef        deadbeef   /

0x70a9adf0:     feedface        feedface   -- REDZONE

0x70a9adf8:     70ae3260        8440c68e   -- Debugging Data
```

```
0x70a9ae00:     5               4ef83     \
0x70a9ae08:     0               0          +- User Data (allocated)
0x70a9ae10:     1               bbddcafe  /
0x70a9ae18:     feedface        139d    -- REDZONE
0x70a9ae20:     70ae3200        d1befaed  -- Debugging Data


0x70a9ae28:     deadbeef        deadbeef  \
0x70a9ae30:     deadbeef        deadbeef   +- User Data (free)
0x70a9ae38:     deadbeef        deadbeef  /
0x70a9ae40:     feedface        feedface  -- REDZONE
0x70a9ae48:     70ae31a0        8440c54e  -- Debugging Data
```

在位于 `0x70a9add8` 和 `0x70a9ae28` 的空闲缓冲区中，redzone 是使用 `0xfeedfacefeedface` 填充的。这是确定缓冲区是否空闲的便利方法。

在从 `0x70a9ae00` 开始的已分配缓冲区中，情况是不同的。请回想一下第 81 页中的 "分配器基础知识"，有两种分配类型：

1) 客户机使用 `kmem_cache_alloc()` 请求的内存，在这种情况下所请求缓冲区的大小等于高速缓存的 bufsize。

2) 客户机使用 `kmem_alloc` (9F) 请求的内存，在这种情况下所请求缓冲区的大小小于或等于高速缓存的 bufsize。 例如，对 20 个字节的请求将从 `kmem_alloc_24` 高速缓存得到满足。 分配器会通过在紧邻客户机数据之后放置一个标记（即 *redzone* **字节**）来强制设置缓冲区边界：

```
0x70a9ae00:     5               4ef83     \
0x70a9ae08:     0               0          +- User Data (allocated)
0x70a9ae10:     1               bbddcafe  /
0x70a9ae18:     feedface        139d    -- REDZONE
0x70a9ae20:     70ae3200        d1befaed  -- Debugging Data
```

位于 `0x70a9ae18` 的 `0xfeedface` 后跟一个 32 位的字，其中包含的内容看似一个随机值。 此数字实际上是缓冲区大小的编码表示形式。 要对此数字进行解码并获得已分配缓冲区的大小，请使用以下公式：

```
size = redzone_value / 251
```

因此，在本示例中，

```
size = 0x139d / 251 = 20 bytes.
```

这表明所请求缓冲区的大小为 20 个字节。 分配器将执行此解码操作，同时会发现 redzone 字节应该位于偏移量为 20 的位置。redzone 字节是十六进制模式 `0xbb`， 如预期的那样存在于 `0x729084e4 (0x729084d0 + 0t20)`。

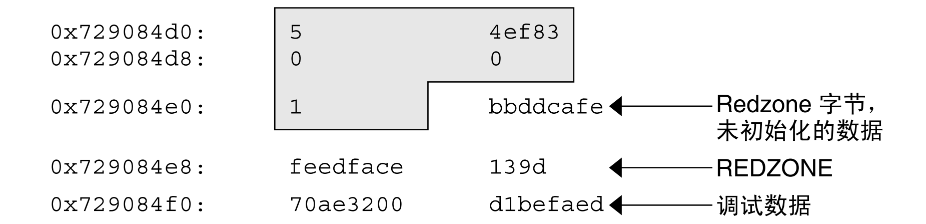


图 9–2 `kmem_alloc(9F)` 缓冲区样例

图 9–3 说明了此内存布局的常规形式。



图 9–3 Redzone 字节

如果分配大小等于高速缓存的 bufsize，则 redzone 字节会覆写 redzone 本身的第一个字节，如图 9–4 中所示。

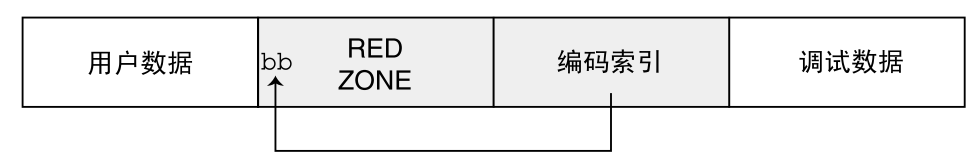


图 9–4 Redzone 开头的 Redzone 字节

此覆写操作会导致 redzone 的第一个 32 位字为 `0xbbedface` 或 `0xfeedfabb`， 具体取决于系统运行的硬件的字节存储顺序。

注 – 为什么分配大小以该方式进行编码？ 要对大小进行编码，分配器可使用公式（251 * 大小 + 1）。对大小进行解码时，整数除法将废弃余数 '+1'。 但是，加上 1 是有价值的，因为分配器可以通过测试（大小 % 251 == 1）是否成立来检查大小是否有效。 这样，分配器可防止损坏 redzone 字节索引。

未初始化的数据：0xbaddcafe

您可能想知道在用 redzone 字节覆写字中的第一个字节**之前**，地址 0x729084d4 上的可疑 0xbbddcafe 是什么。 它是 0xbaddcafe。如果在高速缓存中设置了 KMF_DEADBEEF 标志，则使用 0xbaddcafe 模式填充已分配但**未初始化的**内存。 分配器执行分配时，会循环通过缓冲区的各个字并验证每个字是否包含 0xdeadbeef，然后使用 0xbaddcafe 填充该字。

系统可能会发出以下故障消息：

```
panic[cpu1]/thread=e1979420: BAD TRAP: type=e (Page Fault)

rp=ef641e88 addr=baddcafe occurred in module "unix" due to an

illegal access to a user address
```

在这种情况下，导致故障的地址是 0xbaddcafe： 出现故障的线程访问了一些从未初始化的数据。

将故障消息与失败关联

内核内存分配器会对应于之前所述的失败模式发出故障消息。 例如，系统可能会发出以下故障消息：

```
kernel memory allocator: buffer modified after being freed

modification occurred at offset 0x30
```

由于分配器会尝试验证是否使用 0xdeadbeef 填充了不确定的缓冲区，因此能够检测到此情况。如果偏移的位置为 0x30，则不符合此条件。 由于此条件表明内存损坏，因此分配器会导致系统出现故障。

以下是另一个故障消息示例：

```
kernel memory allocator: redzone violation: write past end of buffer
```

由于分配器会尝试验证 redzone 字节 (0xbb) 是否处于它通过 redzone 大小编码所确定的位置，因此能够检测到此情况。 它无法在正确的位置找到签名字节。 由于这表明内存损坏，因此分配器会导致系统出现紧急情况。 其他的分配器故障消息将在稍后讨论。

内存分配日志记录

本节说明内核内存分配器的日志记录功能以及如何使用它们调试系统崩溃。

buftag 数据完整性

如前所述，每个 buftag 的后半部分包含了有关对应缓冲区的额外信息。此数据中一部分是调试信息，一部分是分配器的专用数据。尽管此辅助数据可以采用几种不同的形式，但是它统称为“缓冲区控制”或 *bufctl* 数据。

不过，分配器需要知道缓冲区的 bufctl 指针是否有效，因为该指针也可能由于异常代码而损坏。分配器通过存储其辅助指针**和**该指针的编码版本，然后交叉检查这两个版本来确认该指针的完整性。

如图 9–5 中所示，这些指针是 *bcp*（缓冲区控制指针）和 *bxstat*（缓冲区控制 XOR 状态）。分配器会对 bcp 和 bxstat 进行排列，以便表达式 `bcp XOR bxstat` 等于已知值。



图 9–5 buftag 中的额外调试数据

这两个指针中的一个或两个损坏时，分配器可以轻易检测到此类损坏，并会导致系统出现紧急情况。**分配**缓冲区后，`bcp XOR bxstat = 0xa110c8ed` ("allocated")。缓冲区空闲时，`bcp XOR bxstat = 0xf4eef4ee` ("freefree")。

注 – 您可能会发现重新检查第 88 页中的 “检查已释放的缓冲区： `0xdeadbeef`”中的示例会有助于确认那里显示的 buftag 指针是否一致。

分配器找到损坏的 buftag 时，会导致系统出现紧急情况，并生成与以下内容类似的消息：

```
kernel memory allocator: boundary tag corrupted

    bcp ^ bxstat = 0xffeef4ee, should be f4eef4ee
```

请记住，如果 bcp 已损坏，仍可通过采用 `bxstat XOR 0xf4eef4ee` 或 `bxstat XOR 0xa110c8ed`（取决于缓冲区是已分配的还是空闲的）的值来对其值进行检索。

`bufctl` 指针

包含在 buftag 区域中的缓冲区控制 (bufctl) 指针可以具有不同的含义，具体取决于高速缓存的 `kmem_flags`。需要特别注意的是，`KMF_AUDIT` 标志的设置情况不同，具体的行为也会有所

不同：如果**未**设置 KMF_AUDIT 标志，则内核内存分配器会为每个缓冲区分配一个 kmem_bufctl_t 结构。此结构包含有关每个缓冲区的一些最少记帐信息。如果**已**设置了 KMF_AUDIT 标志，则分配器会改为分配 kmem_bufctl_audit_t（kmem_bufctl_t 的扩展版本）。

本节假定已设置了 KMF_AUDIT 标志。对于未设置此位的高速缓存，可用的调试信息量会减少。

kmem_bufctl_audit_t（简称 bufctl_audit）包含有关此缓冲区中执行的最后一个事务的其他信息。以下示例说明了如何应用 bufctl_audit 宏检查审计记录。所示的缓冲区是第 88 页中的 “检测内存损坏”中使用的示例缓冲区：

```
> 0x70a9ae00,5/KKn

0x70a9ae00:     5               4ef83

                0               0

                1               bbddcafe

                feedface        139d

                70ae3200        d1befaed
```

使用如上所述的方法可以很容易地看到 0x70ae3200 指向 bufctl_audit 记录：它是 redzone 后面的第一个指针。要检查它所指向的 bufctl_audit 记录，请应用 bufctl_audit 宏：

```
> 0x70ae3200$<bufctl_audit

0x70ae3200:     next            addr            slab

                70378000        70a9ae00        707c86a0

0x70ae320c:     cache           timestamp       thread

                70039928        e1bd0e26afe     70aac4e0

0x70ae321c:     lastlog         contents        stackdepth

                7011c7c0        7018a0b0        4

0x70ae3228:

                kmem_zalloc+0x30

                pid_assign+8

                getproc+0x68
```

```
              cfork+0x60
```

’addr’ 字段是对应于此 bufctl_audit 记录的缓冲区的地址。 以下是原始地址：`0x70a9ae00`。’cache’ 字段是指已分配此缓冲区的 kmem_cache。可以使用 `::kmem_cache` dcmd 对其进行检查，如下所示：

```
> 0x70039928::kmem_cache

ADDR     NAME                      FLAG  CFLAG  BUFSIZE  BUFTOTL

70039928 kmem_alloc_24             020f 000000       24      612
```

’timestamp’ 字段表示执行此事务的时间。此时间的表示方式与 `gethrtime`(3C) 相同。

’thread’ 是指向线程的指针，该线程在此缓冲区中执行了最后一个事务。’lastlog’ 和 ’contents’ 指针指向分配器的**事务日志**中的位置。 这些日志将在第 99 页中的 “分配器日志记录工具 ”中详细讨论。

通常，`bufctl_audit` 提供的最有用的信息段是事务发生时记录的栈跟踪。 在这种情况下，事务是在执行 `fork`(2) 的过程中调用的分配。

高级内存分析

本节介绍用于执行高级内存分析（包括查找内存泄漏和数据损坏原因）的工具。

查找内存泄漏

对于启用了完整 kmem 调试功能集的内核崩溃转储，`::findleaks` dcmd 提供了提供了强大高效的内存泄漏检测。 首次执行 `::findleaks` 时会处理转储中的内存泄漏（这可能需要几分钟）， 然后根据分配栈跟踪合并泄漏。 findleaks 报告会针对所识别的每个内存泄漏显示一个 bufctl 地址和最顶层的栈帧：

```
> ::findleaks

CACHE     LEAKED   BUFCTL CALLER

70039ba8       1 703746c0 pm_autoconfig+0x708

70039ba8       1 703748a0 pm_autoconfig+0x708

7003a028       1 70d3b1a0 sigaddq+0x108

7003c7a8       1 70515200 pm_ioctl+0x187c
```

```
------------------------------------------------------

   Total       4 buffers, 376 bytes
```

使用 bufctl 指针可以通过应用 `bufctl_audit` 宏获取分配的完整栈反向跟踪：

```
> 70d3b1a0$<bufctl_audit

0x70d3b1a0:     next            addr            slab

                70a049c0        70d03b28        70bb7480

0x70d3b1ac:     cache           timestamp       thread

                7003a028        13f7cf63b3      70b38380

0x70d3b1bc:     lastlog         contents        stackdepth

                700d6e60        0               5

0x70d3b1c8:

                kmem_alloc+0x30

                sigaddq+0x108

                sigsendproc+0x210

                sigqkill+0x90

                kill+0x28
```

程序员通常可以使用 `bufctl_audit` 信息和分配栈跟踪快速找到泄漏给定缓冲区的代码路径。

查找数据引用

尝试诊断内存损坏问题时，应该知道其他哪些内核实体包含特定指针的副本。这一点非常重要，因为这可以表明哪个线程在被释放后访问了数据结构。另外，还可以更轻松地了解哪些内核实体正在共享特定（有效）数据项的信息。 `::whatis` 和 `::kgrep` dcmd 可以用于回答这些问题。 可以对相关值应用 `::whatis`：

```
> 0x705d8640::whatis

705d8640 is 705d8640+0, allocated from streams_mblk
```

在本示例中，表明 `0x705d8640` 是指向 STREAMS `mblk` 结构的指针。 要查看整个分配树，请改用 `::whatis -a`：

```
> 0x705d8640::whatis -a

705d8640 is 705d8640+0, allocated from streams_mblk

705d8640 is 705d8000+640, allocated from kmem_va_8192

705d8640 is 705d8000+640 from kmem_default vmem arena

705d8640 is 705d2000+2640 from kmem_va vmem arena

705d8640 is 705d2000+2640 from heap vmem arena
```

这表明分配也会在 kmem_va_8192 高速缓存（即面向 kmem_va vmem 块的 kmem 高速缓存）中进行。 它还显示了 vmem 分配的完整栈。

kmem 高速缓存和 vmem 块的完整列表通过 ::kmastat dcmd 显示。 可以使用 ::kgrep 查找包含指向此 mblk 的指针的其他内核地址。这说明了系统中内存分配的分层性质；通常， 可以根据最具体 kmem 高速缓存的名称确定给定地址所引用的对象类型。

```
> 0x705d8640::kgrep

400a3720

70580d24

7069d7f0

706a37ec

706add34
```

并通过再次应用 ::whatis 对其进行检查：

```
> 400a3720::whatis

400a3720 is in thread 7095b240's stack



> 706add34::whatis

706add34 is 706add20+14, allocated from streams_dblk_120
```

在这里一个指针位于已知内核线程的栈上，另一个指针 mblk 位于对应的 STREAMS dblk 结构的内部。

使用 ::kmem_verify 查找损坏的缓冲区

MDB 的 ::kmem_verify dcmd 在运行时可实现大多数 kmem 分配器实现的检查。可以调用 ::kmem_verify 以便扫描每个具有相应 kmem_flags 的 kmem 内存，或者检查特定的高速缓存。

以下是使用 ::kmem_verify 确定问题的示例：

```
> ::kmem_verify

Cache Name                      Addr     Cache Integrity

kmem_alloc_8                    70039428 clean

kmem_alloc_16                   700396a8 clean

kmem_alloc_24                   70039928 1 corrupt buffer

kmem_alloc_32                   70039ba8 clean

kmem_alloc_40                   7003a028 clean

kmem_alloc_48                   7003a2a8 clean

...
```

在这里可以很容易地看到 kmem_alloc_24 高速缓存包含 ::kmem_verify 所确认的问题。使用显式的高速缓存参数，::kmem_verify dcmd 可以提供有关该问题的更详细信息：

```
> 70039928::kmem_verify

Summary for cache 'kmem_alloc_24'

  buffer 702babc0 (free) seems corrupted, at 702babc0
```

下一步是检查 ::kmem_verify 确认已损坏的缓冲区：

```
> 0x702babc0,5/KKn

0x702babc0:     0               deadbeef

                deadbeef        deadbeef

                deadbeef        deadbeef

                feedface        feedface

                703785a0        84d9714e
```

::kmem_verify 标记此缓冲区的原因此时非常明显：缓冲区中的第一个字（位于 0x702babc0）可能应该使用 0xdeadbeef 模式而不是 0 填充。 此时，为此缓冲区检查 bufctl_audit 可能产生有关最近向缓冲区写入了哪些代码的线索，指明释放此缓冲区的位置和时间。

此情况下的另一种有用方法是使用 ::kgrep 在地址空间中搜索对地址 0x702babc0 的引用，以便发现哪些线程或数据仍然包含对此已释放数据的引用。

分配器日志记录工具

如果为高速缓存设置了 KMF_AUDIT，则内核内存分配器会对记录其最近活动历史的日志进行维护。此**事务日志**记录的是 bufctl_audit 记录。如果同时设置了 KMF_AUDIT 和 KMF_CONTENTS 标志，则分配器会生成一个**内容日志**，其中记录了已分配和已释放缓冲区的部分实际内容。 内容日志的结构和用法不在本文档的讨论范围之内。本节将讨论事务日志。

MDB 提供了用于显示事务日志的多种工具。最简单的工具是 ::walk kmem_log，用于将日志中的事务作为一系列 bufctl_audit_t 指针进行列显：

```
> ::walk kmem_log

70128340

701282e0

70128280

70128220

701281c0

...

> 70128340$<bufctl_audit

0x70128340:     next            addr            slab

                70ac1d40        70bc4ea8        70bb7c00

0x7012834c:     cache           timestamp       thread

                70039428        e1bd7abe721     70aacde0

0x7012835c:     lastlog         contents        stackdepth

                701282e0        7018f340        4

0x70128368:
```

```
                kmem_cache_free+0x24
                nfs3_sync+0x3c
                vfs_sync+0x84
                syssync+4
```

查看整个事务日志的更好方法是使用 ::kmem_log 命令：

```
> ::kmem_log
CPU ADDR     BUFADDR         TIMESTAMP THREAD
  0 70128340 70bc4ea8      e1bd7abe721 70aacde0
  0 701282e0 70bc4ea8      e1bd7aa86fa 70aacde0
  0 70128280 70bc4ea8      e1bd7aa27dd 70aacde0
  0 70128220 70bc4ea8      e1bd7a98a6e 70aacde0
  0 701281c0 70d03738      e1bd7a8e3e0 70aacde0
  ...
  0 70127140 70cf78a0      e1bd78035ad 70aacde0
  0 701270e0 709cf6c0      e1bd6d2573a 40033e60
  0 70127080 70cedf20      e1bd6d1e984 40033e60
  0 70127020 70b09578      e1bd5fc1791 40033e60
  0 70126fc0 70cf78a0      e1bd5fb6b5a 40033e60
  0 70126f60 705ed388      e1bd5fb080d 40033e60
  0 70126f00 705ed388      e1bd551ff73 70aacde0
  ...
```

::kmem_log 的输出按时间标记进行降序排列。ADDR 列是对应于该事务的 bufctl_audit 结构；BUFADDR 指向实际缓冲区。

这些数字表示（分配和释放的）缓冲区中的**事务**。如果特定的缓冲区损坏，则在事务日志中找到该缓冲区，然后确定在其他哪些事务中涉及执行事务的线程可能会有所帮助。这有助于对分配（或释放）缓冲区前后所发生的事件的顺序有一个完整的了解。

可以使用 ::bufctl 命令过滤遍历事务日志的输出。 ::bufctl -a 命令用于按缓冲区地址过滤事务日志中的缓冲区。 以下是对缓冲区 0x70b09578 进行过滤的示例：

```
> ::walk kmem_log | ::bufctl -a 0x70b09578

ADDR     BUFADDR   TIMESTAMP   THREAD   CALLER

70127020 70b09578  e1bd5fc1791 40033e60 biodone+0x108

70126e40 70b09578  e1bd55062da 70aacde0 pageio_setup+0x268

70126de0 70b09578  e1bd52b2317 40033e60 biodone+0x108

70126c00 70b09578  e1bd497ee8e 70aacde0 pageio_setup+0x268

70120480 70b09578  e1bd21c5e2a 70aacde0 elfexec+0x9f0

70120060 70b09578  e1bd20f5ab5 70aacde0 getelfhead+0x100

7011ef20 70b09578  e1bd1e9a1dd 70aacde0 ufs_getpage_miss+0x354

7011d720 70b09578  e1bd1170dc4 70aacde0 pageio_setup+0x268

70117d80 70b09578  e1bcff6ff27 70bc2480 elfexec+0x9f0

70117960 70b09578  e1bcfea4a9f 70bc2480 getelfhead+0x100

...
```

本示例说明一个特定的缓冲区可以在许多事务中使用。

注 – 请记住，kmem 事务日志是内核内存分配器生成的事务不完整记录。会根据需要删除日志中的较旧项，以便保持日志大小不变。

::allocdby 和 ::freedby dcmd 提供了汇总与特定线程关联的事务的便利方法。 以下示例列出了线程 0x70aacde0 最近执行的分配：

```
> 0x70aacde0::allocdby

BUFCTL      TIMESTAMP CALLER

70d4d8c0  e1edb14511a allocb+0x88

70d4e8a0  e1edb142472 dblk_constructor+0xc

70d4a240  e1edb13dd4f allocb+0x88
```

```
70d4e840  e1edb13aeec dblk_constructor+0xc

70d4d860  e1ed8344071 allocb+0x88

70d4e7e0  e1ed8342536 dblk_constructor+0xc

70d4a1e0  e1ed82b3a3c allocb+0x88

70a53f80  e1ed82b0b91 dblk_constructor+0xc

70d4d800  e1e9b663b92 allocb+0x88
```

通过检查 `bufctl_audit` 记录，可以了解特定线程最近的活动。

第 10 章

模块编程 API

本章介绍 MDB 调试器模块 API 中包含的结构和函数。头文件 <sys/mdb_modapi.h> 包含这些函数的原型，SUNWmdbdm 软件包在目录 /usr/demo/mdb 中提供了示例模块的源代码。

调试器模块链接

_mdb_init()

```
const mdb_modinfo_t *_mdb_init(void);
```

为了达到链接和标识目的，要求每个调试器模块提供一个名为 _mdb_init() 的函数。此函数返回一个指向持久性（即，未声明为自动变量）mdb_modinfo_t 结构的指针，如 <sys/mdb_modapi.h> 中所定义：

```
typedef struct mdb_modinfo {

        ushort_t mi_dvers;               /* Debugger API version number */

        const mdb_dcmd_t *mi_dcmds;      /* NULL-terminated list of dcmds */

        const mdb_walker_t *mi_walkers;  /* NULL-terminated list of walks */

} mdb_modinfo_t;
```

mi_dvers 成员用于标识 API 版本号，应该始终设置为 MDB_API_VERSION。因此，当前的版本号会被编译到每个调试器模块中，这样调试器就可以识别和验证模块所使用的应用程序二进制接口。调试器不装入为比调试器本身更新的 API 版本编译的模块。

mi_dcmds 和 *mi_walkers* 成员（如果不为 NULL）分别指向 dcmd 和 walker 定义结构的数组。每个数组都必须以 NULL 元素结尾。这些 dcmd 和 walker 是在模块装入过程中安装并向调试

器注册的。如果一个或多个 dcmd 或 walker 定义不正确，或者它们的名称相冲突或无效，则调试器将拒绝装入模块。Dcmd 和 walker 名称不得包含对调试器具有特殊含义的字符，如引号和圆括号。

模块也可以使用模块 API 执行 `_mdb_init()` 中的代码，以确定它是否适合装入。例如，在存在某些符号时模块只能适合于特定的目标。如果找不到这些符号，则模块可以从 `_mdb_init()` 函数返回 NULL。在这种情况下，调试器将拒绝装入模块，并列显相应的错误消息。

_mdb_fini()

```
void _mdb_fini(void);
```

如果模块在被卸载之前执行某些任务（如释放以前使用 `mdb_alloc()` 分配的持久性内存），则它可以声明一个名为 `_mdb_fini()` 的函数以实现此目的。调试器并不需要此函数。如果已声明，则会在卸载模块之前调用它一次。当用户请求调试器终止时，或者当用户使用 `::unload` 内置 dcmd 显式卸载模块时，将卸载模块。

Dcmd 定义

```
int dcmd(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv);
```

dcmd 是使用类似于 `dcmd()` 声明的函数实现的。此函数接收四个参数并返回整数状态。函数参数如下：

addr 当前地址，也称为点。在 dcmd 的开头，此地址对应于调试器中点 "." 变量的值。

flags 包含以下一个或多个标志的逻辑 OR 的整数：

`DCMD_ADDRSPEC`	在 `::dcmd` 的左侧指定显式地址。
`DCMD_LOOP`	使用 `,count` 语法在循环中调用 dcmd，或者通过管道在循环中调用 dcmd。
`DCMD_LOOPFIRST`	dcmd 函数的此次调用与第一次循环或管道调用相对应。
`DCMD_PIPE`	dcmd 是使用来自于管道的输入调用的。
`DCMD_PIPE_OUT`	使用设置为管道的输出调用 dcmd。

为方便起见，提供了 `DCMD_HDRSPEC()` 宏，这样 dcmd 就可以测试其标志，以确定它是否应该列显标题行（即，它不作为循环的一部分进行调用，或者它作为循环或管道的第一次迭代被调用）。

argc *argv* 数组中的参数数目。

argv 在命令行上 `::dcmd` 右侧指定的参数数组。这些参数可以是字符串，也可以是整数值。

dcmd 函数应返回以下整数值之一，这些整数值是在 `<sys/mdb_modapi.h>` 中定义的。

`DCMD_OK` dcmd 已成功完成。

`DCMD_ERR` dcmd 由于某个原因而失败。

`DCMD_USAGE` 由于指定的参数无效，dcmd 失败。返回此值时，将自动列显 dcmd 用法消息（在下文中描述）。

`DCMD_NEXT` 将使用相同参数自动调用下一个 dcmd 定义（如果存在）。

`DCMD_ABORT` dcmd 失败，而且当前的循环或管道应该异常中止。这与 `DCMD_ERR` 类似，但指示在当前循环或管道中不可能有进一步的进展。

每个 dcmd 都包含一个根据 `dcmd()` 原型示例定义的函数和对应的 `mdb_dcmd_t` 结构，如 `<sys/mdb_modapi.h>` 中所定义。此结构由以下字段组成：

`const char *dc_name` dcmd 的字符串名称，不包含前导的 "`::`"。该名称不能包含任何 MDB 元字符（如 $ 或 ‘）。

`const char *dc_usage` dcmd 的可选用法字符串，在 dcmd 返回 `DCMD_USAGE` 时列显它。例如，如果 dcmd 接受选项 `-a` 和 `-b`，则可能将 `dc_usage` 指定为 "`[-ab]`"。如果 dcmd 不接受任何参数，则可能将 `dc_usage` 设置为 NULL。如果用法字符串以 "`:`" 开头，则说明这是简写形式，指示 dcmd 需要显式地址（即，它要求在其 flags 参数中设置 `DCMD_ADDRSPEC`）。如果用法字符串以 "`?`" 开头，则表明 dcmd 将根据情况决定是否接受地址。这些提示将相应地修改用法消息。

`const char *dc_descr` 一个强制性说明字符串，简要说明 dcmd 的用途。此字符串应仅包含单行文本。

`mdb_dcmd_f *dc_funcp` 指向将被调用以执行 dcmd 的函数的指针。

`void (*dc_help)(void)` 一个指向 dcmd 的 help 函数的可选函数指针。如果此指针不为 NULL，则在用户执行 `::help` *dcmd* 时将调用此函数。此函数可以使用 `mdb_printf()` 显示详细信息或示例。

Walker 定义

```
int walk_init(mdb_walk_state_t *wsp);

int walk_step(mdb_walk_state_t *wsp);

void walk_fini(mdb_walk_state_t *wsp);
```

walker 由 init、step 和 fini 三个函数组成，它们是根据上面的示例原型定义的。 调用 walk 函数（如 mdb_walk()）之一时，或者用户执行 ::walk 内置 dcmd 时，调试器调用 walker。 walk 开始时，MDB 调用 walker 的 init 函数，将新 mdb_walk_state_t 结构的地址传递给该函数，如 <sys/mdb_modapi.h> 中所定义：

```
typedef struct mdb_walk_state {

            mdb_walk_cb_t walk_callback;    /* Callback to issue */

            void *walk_cbdata;              /* Callback private data */

            uintptr_t walk_addr;            /* Current address */

            void *walk_data;                /* Walk private data */

            void *walk_arg;                 /* Walk private argument */

            void *walk_layer;               /* Data from underlying layer */

} mdb_walk_state_t;
```

将会为每个 walk 创建单独的 mdb_walk_state_t，以便同一 walker 的多个实例可以同时处于活动状态。例如，如为 mdb_walk() 指定的那样，状态结构包含 walker 在每个步骤应该调用的回调 (*walk_callback*) 以及回调的专用数据 (*walk_cbdata*)。 *walk_cbdata* 指针对 walker 是不透明的：它既不能修改或取消引用此值，也不能假定它是指向有效内存的指针。

walk 的起始地址存储在 *walk_addr* 中。该地址的值为 NULL（如果调用了 mdb_walk()），或者是为 mdb_pwalk() 指定的地址参数。如果使用内置的 ::walk 命令，则在 ::walk 的左侧指定了显式地址的情况下，*walk_addr* 将不为 NULL。起始地址为 NULL 的 walk 称为**全局** walk。具有显式非 NULL 起始地址的 walk 称为**局部** walk。

示例中的 *walk_data* 和 *walk_arg* 字段用于 walker 的专用存储。 复杂的 walker 可能需要分配辅助状态结构，并将 *walk_data* 设置为指向此结构。 每次启动 walk 时，都会将 *walk_arg* 初始化为对应walker 的 mdb_walker_t 结构的 walk_init_arg 成员的值。

在某些情况下，让几个 walker 共享相同的 init、step 和 fini 例程是很有用的。 例如，MDB genunix 模块为每个内核内存高速缓存提供 walker。 这些 walker 共享相同的 init、step 和 fini 函数，并使用 mdb_walker_t 的 walk_init_arg 成员将相应高速缓存的地址指定为 *walk_arg*。

如果 walker 调用 mdb_layered_walk() 来实例化基础层，则在每次调用 walker 的 step 函数之前，基础层将重置 *walk_addr* 和 *walk_layer*。 基础层将 *walk_addr* 设置为基础对象的目标虚拟地址，并将 *walk_layer* 设置为指向 walker 的局部基础对象副本。 有关分层的 walk 的更多信息，请参阅下面对 mdb_layered_walk() 的讨论。

walker 的 init 和 step 函数应返回以下状态值之一：

`WALK_NEXT`	继续执行下一步。 当 walk init 函数返回 `WALK_NEXT` 时， MDB 调用 walk step 函数。当 walk step 函数返回 `WALK_NEXT` 时，这指示 MDB 应该再次调用 step 函数。
`WALK_DONE`	walk 已成功完成。`WALK_DONE` 可以由 step 函数返回，指示 walk 已完成；也可以由 init 函数返回，指示不需要任何步骤（例如，如果给定的数据结构为空）。
`WALK_ERR`	walk 因出现错误而终止。 如果 `WALK_ERR` 是由 init 函数返回的，则 `mdb_walk()`（或者其任何对应函数）返回 –1 以指示 walker 无法初始化。 如果 `WALK_ERR` 是由 step 函数返回的，则 walk 终止但 `mdb_walk()` 会返回成功信息。

walk_callback 也应返回上述值之一。因此， walk step 函数的任务是确定下一个对象的地址，读入此对象的局部副本， 调用 *walk_callback* 函数， 然后返回其状态。 如果 walk 完成或出现错误，则 step 函数也可以返回 `WALK_DONE` 或 `WALK_ERR`， 而不调用回调。

walker 本身是使用在以下位置中定义的 `mdb_walker_t` 结构定义的：

```
typedef struct mdb_walker {

        const char *walk_name;                /* Walk type name */

        const char *walk_descr;               /* Walk description */

        int (*walk_init)(mdb_walk_state_t *);  /* Walk constructor */

        int (*walk_step)(mdb_walk_state_t *);  /* Walk iterator */

        void (*walk_fini)(mdb_walk_state_t *); /* Walk destructor */

        void *walk_init_arg;                  /* Constructor argument */

} mdb_walker_t;
```

`walk_name` 和 `walk_descr` 字段应分别初始化为指向包含 walker 的名称和简短说明的字符串。要求 walker 具有非 NULL 名称和说明，而且该名称不能包含任何 MDB 元字符。 说明字符串由 `::walkers` 和 `::dmods` 内置 dcmd 列显。

`walk_init`、`walk_step` 和 `walk_fini` 成员引用 walk 函数本身， 如前所述。可以将 `walk_init` 和 `walk_fini` 成员设置为 NULL，以指示不需要执行特殊的初始化或清除操作。不能将 `walk_step` 成员设置为 NULL。如前所述，`walk_init_arg` 成员用于初始化为给定 walker 创建的每个新 `mdb_walk_state_t` 的 `walk_arg` 成员。图 10–1 显示典型 walker 的算法的流程图。

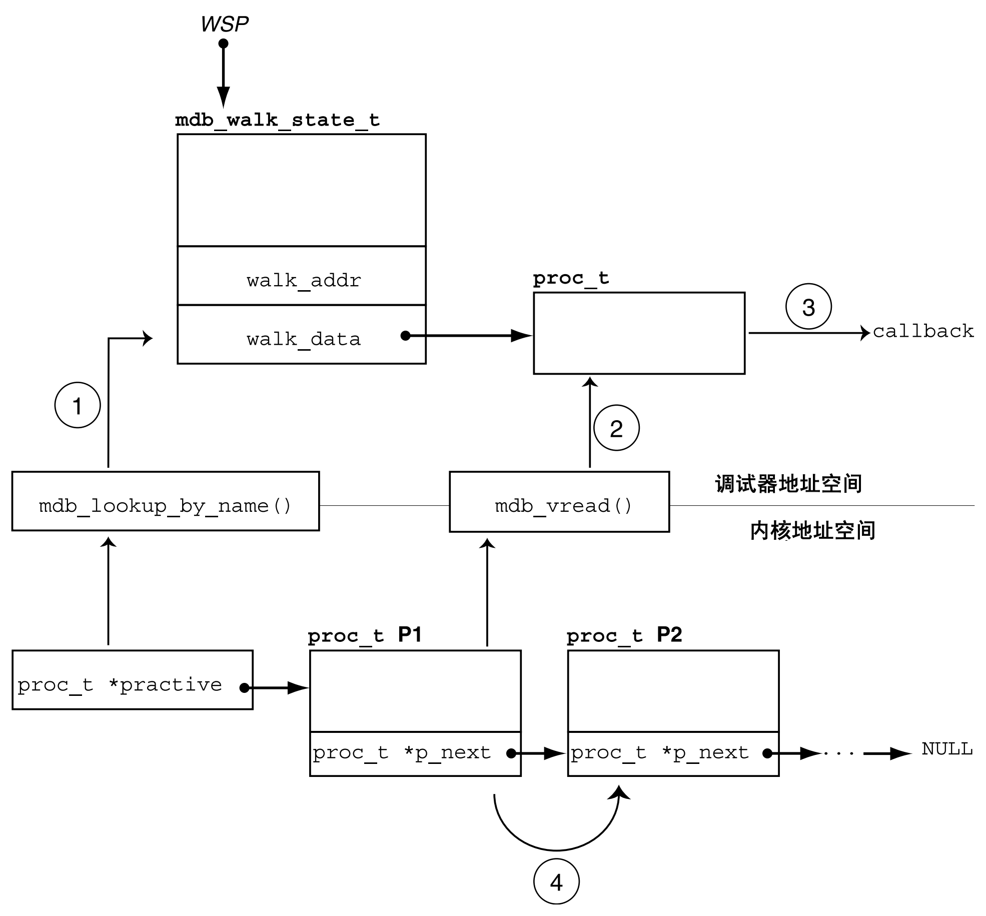


图 10–1 Walker 样例

walker 设计用于迭代内核中 `proc_t` 结构的列表。列表头存储在全局 `practive` 变量中，每个元素的 `p_next` 指针都指向列表中的下一个 `proc_t`。该列表以 NULL 指针结尾。在 walker 的 `init` 例程中，使用 `mdb_lookup_by_name()` step (1) 查找 `practive` 符号，然后将其值复制到 *wsp* 指向的 `mdb_walk_state_t` 中。

在 walker 的 step 函数中，使用 `mdb_vread()` step (2) 将列表中的下一个 `proc_t` 结构复制到调试器的地址空间中，使用指向此局部副本的指针 step (3) 调用回调函数，然后在下一次迭代中使用 `proc_t` 结构的地址更新 `mdb_walk_state_t`。此更新对应于指向列表中下一个元素的指针 step (4)。

这些 step 说明了典型 walker 的结构：init 例程查找特定数据结构的全局信息，step 函数将其读入下一个数据项的局部副本，然后将其传递到回调函数，读取下一个元素的地址。最后，在 walk 终止时，fini 函数释放任何专用存储。

API 函数

mdb_pwalk()

```
int mdb_pwalk(const char *name, mdb_walk_cb_t func, void *data,

              uintptr_t addr);
```

使用 *name* 指定的 walker 启动从 *addr* 开始的局部 walk，然后在每一步调用回调函数 *func*。如果 *addr* 为 NULL，则执行全局 walk（即，`mdb_pwalk()` 调用相当于对没有结尾 *addr* 参数的 `mdb_walk()` 的调用）。 如果成功，此函数返回 0；如果出错，此函数返回 -1。 如果 walker 本身返回致命错误，或者如果调试器无法识别指定的 walker 名称，则 `mdb_pwalk()` 函数失败。如果存在命名冲突，则可以使用反引号 (‘) 运算符限定 walker 名称的作用域。*data* 参数是仅对调用方有意义的不透明参数；在 walk 的每一步中都将它传递回 *func*。

mdb_walk()

```
int mdb_walk(const char *name, mdb_walk_cb_t func, void *data);
```

使用 *name* 指定的 walker 启动从 *addr* 开始的全局 walk，然后在每一步调用回调函数 *func*。如果成功，此函数返回 0；如果出错，此函数返回 -1。 如果 walker 本身返回致命错误，或者如果调试器无法识别指定的 walker 名称，则 `mdb_walk()` 函数失败。如果存在命名冲突，则可以使用反引号 (‘) 运算符限定 walker 名称的作用域。data 参数是仅对调用方有意义的不透明参数；在 walk 的每一步中都将它传递回 *func*。

mdb_pwalk_dcmd()

```
int mdb_pwalk_dcmd(const char *wname, const char *dcname, int argc,

            const mdb_arg_t *argv, uintptr_t addr);
```

使用 *wname* 指定的 walker 启动从 *addr* 开始的局部 walk，然后使用指定的 *argc* 和 *argv* 在每一步调用由 *dcname* 指定的 dcmd。如果成功，此函数返回 0；如果出错，此函数返回 -1。 如果 walker 本身返回致命错误，如果调试器无法识别指定的 walker 名称或 dcmd 名称，或者如果 dcmd 本身将 `DCMD_ABORT` 或 `DCMD_USAGE` 返回给 walker，则该函数失败。如果存在命名冲突，则可以使用反引号 (‘) 运算符限定 walker 名称和 dcmd 名称的作用域。 从 `mdb_pwalk_dcmd()` 调用时， dcmd 将在其 flags 参数中设置 `DCMD_LOOP` 和 `DCMD_ADDRSPEC` 位，而且首次调用时将设置 `DCMD_LOOPFIRST`。

mdb_walk_dcmd()

```
int mdb_walk_dcmd(const char *wname, const char *dcname, int argc,

            const mdb_arg_t *argv);
```

使用 *wname* 指定的 walker 启动全局 walk，然后使用指定的 *argc* 和 *argv* 在每一步调用由 *dcname* 指定的 dcmd。 如果成功，此函数返回 0；如果出错，此函数返回 -1。如果 walker 本身返回致命错误，如果调试器无法识别指定的 walker 名称或 dcmd 名称，或者如果 dcmd 本身将 DCMD_ABORT 或 DCMD_USAGE 返回给 walker，则该函数失败。如果存在命名冲突，则可以使用反引号 (‘) 运算符限定 walker 名称和 dcmd 名称的作用域。 从 mdb_walk_dcmd() 调用时，dcmd 将在其 flags 参数中设置 DCMD_LOOP 和 DCMD_ADDRSPEC 位，而且首次调用时将设置 DCMD_LOOPFIRST。

mdb_call_dcmd()

```
int mdb_call_dcmd(const char *name, uintptr_t addr, uint_t flags,

            int argc, const mdb_arg_t *argv);
```

使用给定的参数调用指定的 dcmd 名称。 将点变量重置为 *addr*，并将 *addr*、*flags*、*argc* 和 *argv* 传递到 dcmd。 如果成功，此函数返回 0；如果出错，此函数返回 -1。如果 dcmd 返回 DCMD_ERR、DCMD_ABORT 或 DCMD_USAGE，或者如果调试器无法识别指定的 dcmd 名称，则该函数失败。 如果存在命名冲突，则可以使用反引号 (‘) 运算符限定 dcmd 名称的作用域。

mdb_layered_walk()

```
int mdb_layered_walk(const char *name, mdb_walk_state_t *wsp);
```

在使用指定 walker *name* 启动的 walk 的顶部，对由 *wsp* 表示的 walk 进行分层。如果存在命名冲突，则可以使用反引号 (‘) 运算符限定名称的作用域。例如，可以使用分层的 walk，以便协助为在其他数据结构中嵌入的数据结构构造 walker。

例如，假定内核中的每个 CPU 结构都包含一个指向嵌入式结构的指针。要为嵌入式结构类型编写 walker，您可以复制代码以迭代 CPU 结构和取消引用每个 CPU 结构的相应成员，或者可以在现有 CPU walker 的顶部对嵌入式结构的 walker 进行分层。

mdb_layered_walk() 函数从 walker 的 init 例程内使用，用于将新层添加到当前 walk。 基础层是在调用 mdb_layered_walk() 的过程中初始化的。调用 walk 例程会传入指向其当前 walk 状态的指针；此状态用于构造分层 walk。在调用调用方的 walk fini 函数后，将清除每个分层 walk。 如果将多个层添加到 walk，则调用方的 walk step 函数将逐步通过第一层返回的每个元素，然后是第二层返回的每个元素，依此类推。

如果成功，`mdb_layered_walk()` 函数返回 0；如果出错，`mdb_layered_walk()` 函数返回 -1。如果调试器无法识别指定的 walker 名称，如果 *wsp* 指针不是有效的、处于活动 walk 状态的指针，如果分层 walker 本身无法初始化，或者如果调用方尝试在其自身的顶部对 walker 进行分层，则该函数失败。

mdb_add_walker()

```
int mdb_add_walker(const mdb_walker_t *w);
```

向调试器注册新的 walker。根据第 25 页中的 "dcmd 和 Walker 名称解析"中所述的名称解析规则，将 walker 添加到模块的名称空间和调试器的全局名称空间。如果成功，此函数返回 0；如果由于此模块已注册给定的 walker 名称或 walker 结构 *w* 未正确构造而出错，此函数返回 -1。`mdb_walker_t` *w* 中的信息将复制到内部调试器结构，因此在调用 `mdb_add_walker()` 后调用方可以重新使用或释放此结构。

mdb_remove_walker()

```
int mdb_remove_walker(const char *name);
```

删除具有指定 *name* 的 walker。如果成功，此函数返回 0；如果出错，此函数返回 -1。将从当前模块的名称空间中删除 walker。如果 walker 名称是未知的，或者它仅在另一模块的名称空间中注册，则该函数失败。`mdb_remove_walker()` 函数可以用于删除使用 `mdb_add_walker()` 动态添加的 walker，或者以静态方式作为模块链接结构的一部分添加的 walker。在 walker 名称中不能使用作用域运算符；`mdb_remove_walker()` 的调用方尝试删除其他模块导出的 walker 是不合法的。

mdb_vread() 和 mdb_vwrite()

```
ssize_t mdb_vread(void *buf, size_t nbytes, uintptr_t addr);
```

```
ssize_t mdb_vwrite(const void *buf, size_t nbytes, uintptr_t addr);
```

使用这些函数可以从给定的目标虚拟地址（由 *addr* 参数指定）读取和写入数据。`mdb_vread()` 函数在读取成功时返回 *nbytes*，出错时返回 -1；如果读取由于只能从指定的地址读取一部分数据而被截断，则返回 -1。`mdb_vwrite()` 函数在写入成功时返回实际写入的字节数；出错时返回 -1。

mdb_fread() 和 mdb_fwrite()

```
ssize_t mdb_fread(void *buf, size_t nbytes, uintptr_t addr);
```

```
ssize_t mdb_fwrite(const void *buf, size_t nbytes, uintptr_t addr);
```

使用这些函数可以从对应于给定的目标虚拟地址（由 addr 参数指定）的目标文件位置读取和写入数据。mdb_fread() 函数在读取成功时返回 nbytes，出错时返回 -1；如果读取由于只能从指定的地址读取一部分数据而被截断，则返回 -1。mdb_fwrite() 函数在写入成功时返回实际写入的字节数；出错时返回 -1。

mdb_pread() 和 mdb_pwrite()

```
ssize_t mdb_pread(void *buf, size_t nbytes, uint64_t addr);
```

```
ssize_t mdb_pwrite(const void *buf, size_t nbytes, uint64_t addr);
```

使用这些函数可以从给定的目标物理地址（由 *addr* 参数指定）读取和写入数据。mdb_pread() 函数在读取成功时返回 *nbytes*，出错时返回 -1；如果读取由于只能从指定的地址读取一部分数据而被截断，则返回 -1。mdb_pwrite() 函数在写入成功时返回实际写入的字节数；出错时返回 -1。

mdb_readstr()

```
ssize_t mdb_readstr(char *s, size_t nbytes, uintptr_t addr);
```

mdb_readstr() 函数将从目标虚拟地址 *addr* 开始、以空字符结尾的 C 字符串读入由 *s* 寻址的缓冲区。该缓冲区的大小由 *nbytes* 指定。如果字符串过长而无法放在缓冲区中，则会将字符串截断到缓冲区大小，并在 s[nbytes - 1] 处存储空字节。成功时将返回在 *s* 中存储的字符串长度（不包括结尾的空字节）；否则返回 -1 以指示出现了错误。

mdb_writestr()

```
ssize_t mdb_writestr(const char *s, uintptr_t addr);
```

mdb_writestr() 函数将 *s* 中以空字符结尾的 C 字符串（包括结尾的空字节）写入 *addr* 所指定地址处的目标虚拟地址空间。成功时将返回写入的字节数（不包括结尾的空字节）；否则返回 -1 以指示出现了错误。

mdb_readsym()

```
ssize_t mdb_readsym(void *buf, size_t nbytes, const char *name);
```

mdb_readsym() 与 mdb_vread() 类似，只不过是读取开始的虚拟地址是根据 *name* 指定的符号的值获取的。如果找不到该名称的符号或者出现读取错误，则返回 -1；否则返回 *nbytes* 以表示成功。

如果需要区分符号查找失败和读取失败，则调用方可以首先单独查找符号。主要可执行文件的符号表用于符号查找；如果符号驻留在其他符号表中，则必须首先应用 mdb_lookup_by_obj()，然后应用 mdb_vread()。

mdb_writesym()

```
ssize_t mdb_writesym(const void *buf, size_t nbytes, const char *name);
```

mdb_writesym() 与 mdb_vwrite() 相同，只不过写入开始的虚拟地址是根据按名称指定的符号的值获取的。如果找不到该名称的符号，则返回 -1。否则，在成功时返回成功写入的字节数，出错时返回 -1。主要可执行文件的符号表用于符号查找；如果符号驻留在其他符号表中，则必须首先应用 mdb_lookup_by_obj()，然后应用 mdb_vwrite()。

mdb_readvar() 和 mdb_writevar()

```
ssize_t mdb_readvar(void *buf, const char *name);

ssize_t mdb_writevar(const void *buf, const char *name);
```

mdb_readvar() 与 mdb_vread() 类似，只不过读取开始的虚拟地址和要读取的字节数是根据 *name* 指定的符号的值和大小获取的。 如果找不到该名称的符号，则返回 -1。成功时返回符号大小（读取的字节数）；出错时返回 -1。对于读取具有固定大小的已知变量，这是很有用的。例如：

```
int hz;     /* system clock rate */

mdb_readvar(&hz, "hz");
```

如果需要区分符号查找失败和读取失败，则调用方可以首先单独查找符号。 调用方还必须仔细检查相关符号的定义，以便确保局部声明的类型与目标定义的类型完全相同。例如，如果调用方声明 int，相关符号实际上是 long，而且调试器检查的是 64 位内核目标，则 mdb_readvar() 会将 8 个字节复制回调用方的缓冲区，在存储 int 后损坏 4 个字节。

mdb_writevar() 与 mdb_vwrite() 相同，只不过写入开始的虚拟地址和要写入的字节数是根据按名称指定的符号的值和大小获取的。 如果找不到该名称的符号，则返回 -1。否则，在成功时返回成功写入的字节数，出错时返回 -1。

对于这两个函数，主要可执行文件的符号表用于符号查找；如果符号驻留在其他符号表中，则必须首先应用 mdb_lookup_by_obj()，然后应用 mdb_vread() 或 mdb_vwrite()。

mdb_lookup_by_name() 和 mdb_lookup_by_obj()

```
int mdb_lookup_by_name(const char *name, GElf_Sym *sym);

int mdb_lookup_by_obj(const char *object, const char *name, GElf_Sym *sym);
```

查找指定的符号名称，并将 ELF 符号信息复制到 *sym* 指向的 `GElf_Sym` 中。如果找到符号，则函数返回 0；否则返回 -1。*name* 参数指定符号名称。*object* 参数通知调试器查找符号的位置。对于 `mdb_lookup_by_name()` 函数，目标文件缺省为 `MDB_OBJ_EXEC`。对于 `mdb_lookup_by_obj()`，对象名称应该为以下名称之一：

`MDB_OBJ_EXEC`	在可执行文件的符号表（`.symtab` 部分）中查找。对于内核崩溃转储，这对应于 `unix.X` 文件或 `/dev/ksyms` 中的符号表。
`MDB_OBJ_RTLD`	在运行时链接编辑器的符号表中查找。对于内核崩溃转储，这对应于 `krtld` 模块的符号表。
`MDB_OBJ_EVERY`	在所有的已知符号表中查找。对于内核崩溃转储，这包括 `unix.X` 文件或 `/dev/ksyms` 中的 `.symtab` 和 `.dynsym` 部分，以及每个模块符号表（如果已处理它们）。
`object`	如果显式指定了特定装入对象的名称，则仅搜索此对象的符号表。可以根据第 24 页中的 "符号名称解析"中所述的装入对象命名约定来命名对象。

mdb_lookup_by_addr()

```
int mdb_lookup_by_addr(uintptr_t addr, uint_t flag, char *buf,

                size_t len, GElf_Sym *sym);
```

查找与指定地址相对应的符号，然后将 ELF 符号信息复制到 *sym* 指向的 `GElf_Sym` 中，将符号名称复制到 *buf* 寻址的字符数组中。如果找到对应符号，则函数返回 0；否则返回 -1。

标志参数指定查找模式，应为以下参数之一：

`MDB_SYM_FUZZY`	允许基于当前的符号距离设置进行模糊匹配。可以使用内置的 `::set -s` 命令控制符号距离。如果已设置显式符号距离（绝对模式），则在符号值与地址的距离不超过绝对符号距离时，地址可以与符号匹配。如果启用智能模式（符号距离 = 0），则在地址位于范围 [符号值, 符号值 + 符号大小) 之内时，它可以与符号匹配。
`MDB_SYM_EXACT`	不允许模糊匹配。仅当符号值正好等于指定地址时，符号才能与地址匹配。

如果出现符号匹配，则将符号的名称复制到调用方提供的 *buf* 中。*len* 参数指定此缓冲区的长度（以字节为单位）。调用方的 *buf* 的大小至少应为 MDB_SYM_NAMLEN 字节。调试器将名称复制到此缓冲区，并附加结尾空字节。如果名称长度超过了缓冲区的长度，则该名称将被截断，但是它始终包括结尾空字节。

mdb_getopts()

```
int mdb_getopts(int argc, const mdb_arg_t *argv, ...);
```

解析和处理指定参数数组 (*argv*) 中的选项和选项参数。 *argc* 参数表示参数数组的长度。此函数按顺序处理每个参数，然后停止和返回无法处理的第一个参数的数组索引。 如果成功处理了所有参数，则返回 *argc*。

在 *argc* 和 *argv* 参数后面，mdb_getopts() 函数接受参数（说明应出现在 *argv* 数组中的选项）的变量列表。 每个选项都由一个选项字母（char 参数）、一种选项类型（uint_t 参数）以及一个或两个附加参数来说明，如下表所示。选项参数列表以 NULL 参数结尾。 类型应为以下类型之一：

MDB_OPT_SETBITS

该选项将指定的位 OR 到标志字中。该选项由以下参数说明：

```
char c, uint_t type, uint_t bits, uint_t *p
```

如果类型为 MDB_OPT_SETBITS，而且在 *argv* 列表中检测到选项 *c*，则调试器将位 OR 到指针 *p* 引用的整数中。

MDB_OPT_CLRBITS

该选项从标志字中清除指定的位。该选项由以下参数说明：

```
char c, uint_t type, uint_t bits, uint_t *p
```

如果类型为 MDB_OPT_CLRBITS，而且在 *argv* 列表中检测到选项 *c*，则调试器将从指针 *p* 引用的整数中清除位。

MDB_OPT_STR

该选项接受字符串参数。该选项由以下参数说明：

```
char c, uint_t type, const char **p
```

如果类型为 MDB_OPT_STR，而且在 *argv* 列表中检测到选项 *c*，则调试器在 *p* 引用的指针中存储指向 *c* 后面的字符串参数的指针。

MDB_OPT_UINTPTR

该选项接受 uintptr_t 参数。该选项由以下参数说明：

```
char c, uint_t type, uintptr_t *p
```

如果类型为 MDB_OPT_UINTPTR，而且在 *argv* 列表中检测到选项 c，则调试器在 *p* 引用的 uintptr_t 中存储 *c* 后面的整数参数。

MDB_OPT_UINT64 该选项接受 uint64_t 参数。该选项由以下参数说明：

char c, uint_t type, uint64_t *p

如果类型为 MDB_OPT_UINT64，而且在 *argv* 列表中检测到选项 *c*，则调试器在 *p* 引用的 uint64_t 中存储 *c* 后面的整数参数。

例如，以下源代码：

```
int

dcmd(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)

{

        uint_t opt_v = FALSE;

        const char *opt_s = NULL;



        if (mdb_getopts(argc, argv,

            ’v’, MDB_OPT_SETBITS, TRUE, &opt_v,

            ’s’, MDB_OPT_STR, &opt_s, NULL) != argc)

                return (DCMD_USAGE);



        /* ... */

}
```

说明如何在 dcmd 中使用 mdb_getopts()，以接受布尔选项 "-v"（它将 *opt_v* 变量设置为 TRUE）和选项 "-s"（它接受存储在 *opt_s* 变量中的字符串参数）。如果在返回给调用方之前，mdb_getopts() 函数检测到选项字母无效或者缺少选项参数，则它还会自动发出警告消息。在 dcmd 完成时，调试器将自动对参数字符串和 *argv* 数组的存储进行垃圾收集。

mdb_strtoull()

```
u_longlong_t mdb_strtoull(const char *s);
```

将指定的字符串 *s* 转换为 `unsigned long long` 表示形式。此函数适用于处理和转换不适合使用 `mdb_getopts()` 进行处理和转换的字符串参数。如果无法将字符串参数转换为有效的整数表示形式，则该函数失败，同时列显相应的错误消息并异常中止 dcmd。因此，不需要错误检查代码。字符串可以使用任何有效基数说明符（0i、0I、0o、0O、0t、0T、0x 或 0X）作为前缀；否则，使用缺省基数解释它。如果 *s* 中的任何字符不适合于基数，或者如果发生整数溢出，则该函数将失败并异常中止 dcmd。

mdb_alloc()、mdb_zalloc() 和 mdb_free()

```
void *mdb_alloc(size_t size, uint_t flags);
```

```
void *mdb_zalloc(size_t size, uint_t flags);
```

```
void mdb_free(void *buf, size_t size);
```

`mdb_alloc()` 分配 *size* 字节的调试器内存，并返回指向已分配内存的指针。已分配的内存至少是双字对齐的，以便它可以保存任何 C 数据结构。 不能采用更高的对齐方式。*flags* 参数应该是以下一个或多个值的按位 `OR`：

`UM_NOSLEEP` 如果完成请求所需的足够内存不是立即可用，则返回 NULL 以指示失败。调用方必须检查 NULL 并相应处理此情况。

`UM_SLEEP` 如果完成请求所需的足够内存不是立即可用，则在可以完成请求之前一直休眠。因此，`UM_SLEEP` 分配一定会成功。调用方无需检查 NULL 返回值。

`UM_GC` 在此调试器命令结束时， 自动对分配进行垃圾收集。调用方随后不应在此块上调用 `mdb_free()`， 因为调试器将自动执行取消分配操作。 从 dcmd 内进行的所有内存分配都必须使用 `UM_GC`， 以便在用户中断 dcmd 的情况下， 调试器可以对内存进行垃圾收集。

`mdb_zalloc()` 与 `mdb_alloc()` 类似，但是将已分配内存返回给调用方之前用零填充该内存。对于 `mdb_alloc()` 所返回内存的初始内容没有任何保证。`mdb_free()` 用于释放以前分配的内存（除非为它分配了 `UM_GC`）。 缓冲区地址和大小必须与原始分配完全匹配。通过 `mdb_free()` 仅释放部分分配是不合法的。多次释放分配也是不合法的。零字节分配始终返回 NULL；释放大小为零的 NULL 指针总是会成功。

mdb_printf()

```
void mdb_printf(const char *format, ...);
```

使用指定的格式字符串和参数列显带格式的输出。模块编写者应该将 `mdb_printf()` 用于所有输出，但警告和错误消息除外。在适当的情况下，此函数会自动触发内置的输出页面调度程序。`mdb_printf()` 函数与 `printf`(3C) 类似，但有一些例外：不支持宽字符串的 `%C`、`%S`

和 %ws 说明符，不支持 %f 浮点格式，两种可能的双精度格式的 %e、%E、%g 和 %G 说明符仅生成单一样式的输出，不支持 %.n 格式的精度说明。支持的说明符列表如下：

标志说明符

%# 如果在格式字符串中找到 # 符号，则这会选择给定格式的替换形式。 并不是所有的格式都具有替换形式；替换形式随格式的不同而不同。有关替换格式的详细信息，请参阅下面的格式说明。

%+ 列显带符号值时，始终显示符号（即以 '+' 或 '-' 为前缀）。 如果没有 %+，则正值不带有符号前缀，而在负值前面加上 '-' 前缀。

%- 在指定的字段宽度内左对齐输出。如果输出的宽度小于指定的字段宽度，则在右侧用空白填充输出。如果没有 %-，则缺省情况下将值右对齐。

%0 如果输出是右对齐的，而且输出的宽度小于指定的字段宽度，则用零填充输出字段。如果没有 %0，则在右对齐的值前面加上空白，以填充字段。

字段宽度说明符

%n 将字段宽度设置为指定的十进制值。

%? 将字段宽度设置为十六进制指针值的最大宽度。在 ILP32 环境中为 8， 在 LP64 环境中为 16。

%* 将字段宽度设置为在参数列表的当前位置指定的值。假定此值为 int。请注意，在 64 位编译环境中， 可能需要将 long 值强制转换为 int。

整数说明符

%h 要列显的整数值为 short。

%l 要列显的整数值为 long。

%ll 要列显的整数值为 long long。

终端属性说明符

如果调试器的标准输出是终端， 而且可以通过 terminfo 数据库获取终端属性，则可以使用以下终端转义结构：

%<n> 启用对应于 *n* 的终端属性。对于 %<> 的每个实例， 只能启用一个属性。

%</n> 禁用对应于 *n* 的终端属性。请注意， 在反白显示、灰显文本和粗体文本的情况下，禁用这些属性的终端代码可能是相同的。因此，相互独立地禁用这些属性也许是不可能的。

如果没有可用的终端信息，则 mdb_printf() 将忽略每个终端属性结构。有关终端属性的更多信息，请参见 terminfo(4)。可用的 terminfo 属性如下：

a 替换字符集

b	粗体文本
d	灰显文本
r	反白显示
s	最突出的功能
u	加下划线

格式说明符

%% 列显 '%' 符号。

%a 以符号形式列显地址。与 %a 关联的值的最小大小是 uintptr_t；不需要指定 %la。如果打开地址到符号的转换，则调试器会尝试将地址转换为符号名称后跟当前输出中的偏移，并列显此字符串；否则，在缺省输出基数中列显值。 如果使用 %#a，则替换格式会将 ':' 后缀添加到输出。

%A 此格式与 %a 相同，只不过无法将地址转换为符号名称以及偏移时，不列显任何内容。如果使用 %#A，则在地址转换失败时替换格式列显 '?'。

%b 以符号形式解码并列显位字段。此说明符需要两个连续参数：位字段值（对于 %b 为 int， 对于 %lb 为 long， 等等）和一个指向 mdb_bitmask_t 结构的数组的指针：

```
typedef struct mdb_bitmask {

        const char *bm_name;       /* String name to print */

        u_longlong_t bm_mask;      /* Mask for bits */

        u_longlong_t bm_bits;      /* Result for value & mask */

} mdb_bitmask_t;
```

数组应该以其 bm_name 字段设置为 NULL 的结构结尾。使用 %b 时， 调试器读取值参数，然后循环检查每个 mdb_bitmask 结构以查看是否为以下情况：

```
(value & bitmask->bm_mask) == bitmask->bm_bits
```

如果此表达式为真，则列显 bm_name 字符串。列显的每个字符串由逗号分隔。以下示例说明如何使用 %b 解码 kthread_t 中的 t_flag 字段：

```
const mdb_bitmask_t t_flag_bits[] = {

        { "T_INTR_THREAD", T_INTR_THREAD, T_INTR_THREAD },

        { "T_WAKEABLE", T_WAKEABLE, T_WAKEABLE },
```

```
        { "T_TOMASK", T_TOMASK, T_TOMASK },

        { "T_TALLOCSTK", T_TALLOCSTK, T_TALLOCSTK },

            ...

        { NULL, 0, 0 }

};



void

thr_dump(kthread_t *t)

{

        mdb_printf("t_flag = <%hb>\n", t->t_flag, t_flag_bits);



        ...

}
```

如果 `t_flag` 设置为 0x000a，则该函数将列显：

```
t_flag = <T_WAKEABLE,T_TALLOCSTK>
```

如果指定 `%#b`，则会将与位掩码数组中的元素不匹配的所有位的联合列显为已解码名称后面的十六进制值。

`%c` 将指定的整数列显为 ASCII 字符。

`%d` 将指定的整数列显为带符号十进制值。与 `%i` 相同。如果指定 `%#d`，则替换格式将为值添加前缀 '`0t`'。

`%e` 以浮点格式 `[+/-]d.ddddddde[+/-]dd` 列显指定的双精度数，其中在基数字符前有一位，精度为七位，在指数后至少有两位。

`%E` 使用与 `%e` 相同的规则列显指定的双精度数，但指数字符将是 '`E`' 而不是 '`e`'。

`%g` 以与 `%e` 相同的浮点格式列显指定的双精度数，但精度为十六位。如果指定 `%llg`，则参数应为 `long double` 类型（四精度浮点值）。

`%G` 使用与 `%g` 相同的规则列显指定的双精度数，但指数字符将是 '`E`' 而不是 '`e`'。

`%i` 将指定的整数列显为带符号十进制值。与 `%d` 相同。如果指定 `%#i`，则替换格式将为值添加前缀 '`0t`'。

%I	将指定的 32 位无符号整数列显为点分十进制格式的 Internet IPv4 地址（例如，十六进制值 0xffffffff 将列显为 255.255.255.255）。
%m	列显由空格组成的边距。如果未指定字段，则使用缺省的输出边距宽度；否则，字段宽度确定列显的空格字符数。
%o	将指定的整数列显为无符号八进制值。如果使用 %#o，则替换格式将为输出添加前缀 '0'。
%p	将指定的指针 (void *) 列显为十六进制值。
%q	将指定的整数列显为带符号八进制值。如果使用 %#o，则替换格式将为输出添加前缀 '0'。
%r	在当前输出基数中将指定的整数列显为无符号值。用户可以使用 $d dcmd 更改输出基数。如果指定 %#r，则替换格式将为值添加相应的基数前缀：'0i'（用于二进制）、'0o'（用于八进制）、'0t'（用于十进制）或 '0x'（用于十六进制）。
%R	在当前输出基数中将指定的整数列显为带符号值。如果指定 %#R，则替换格式将为值添加相应的基数前缀。
%s	列显指定的字符串 (char *)。如果字符串指针为 NULL，则列显字符串 '<NULL>'。
%t	前移一个或多个制表停止位置。如果未指定宽度，则输出前移到下一个制表停止位置；否则，字段宽度确定前移多少个制表停止位置。
%T	将输出列前移到字段宽度的下一个倍数。如果未指定字段宽度，则不执行操作。如果当前输出列不是字段宽度的倍数，则将添加空格以前移输出列。
%u	将指定的整数列显为无符号十进制值。如果指定 %#u，则替换格式将为值添加前缀 '0t'。
%x	将指定的整数列显为十六进制值。字符 a-f 用作代表值 10-15 的符号。如果指定 %#x，则替换格式将为值添加前缀 '0x'。
%X	将指定的整数列显为十六进制值。字符 A-F 用作值 10-15 的位。如果指定 %#X，则替换格式将为值添加前缀 '0X'。
%Y	将指定的 time_t 列显为字符串 'year month day HH:MM:SS'。

mdb_snprintf()

```
size_t mdb_snprintf(char *buf, size_t len, const char *format, ...);
```

基于指定的格式字符串和参数构造带格式字符串，并将生成的字符串存储在指定的 *buf* 中。mdb_snprintf() 函数接受与 mdb_printf() 函数相同的格式说明符和参数。*len* 参数指定 *buf* 的大小（以字节为单位）。 在 *buf* 中放置的带格式字节不超过 *len* - 1 个；mdb_snprintf() 始终使 *buf* 以空字节结尾。该函数返回完整的带格式字符串所需的字节数，不包括结尾的空字

节。如果 *buf* 参数为 NULL，而且 *len* 设置为零，则该函数不会将任何字符存储到 *buf* 中，但返回完整的带格式字符串所需的字节数；此技术可用于确定适当的缓冲区大小以进行动态内存分配。

mdb_warn()

```
void mdb_warn(const char *format, ...);
```

将错误或警告消息列显到标准错误。`mdb_warn()` 函数接受格式字符串和可能包含为 `mdb_printf()` 记录的任一说明符的变量参数列表。但是，`mdb_warn()` 的输出被发送到标准错误，未进行缓冲，且未通过输出页面调度程序发送或作为 dcmd 管道的一部分处理。 所有错误消息都自动使用字符串 "`mdb:`" 作为前缀。

此外，如果 *format* 参数不包含换行符 (`\n`)，则格式字符串将隐式带有后缀字符串 "`:%s\n`"，其中 `%s` 被替换为对应于模块 API 函数记录的最后一个错误的错误消息字符串。例如，以下源代码：

```
if (mdb_lookup_by_name("no_such_symbol", &sym) == -1)

       mdb_warn("lookup_by_name failed");
```

生成以下输出：

```
mdb: lookup_by_name failed: unknown symbol name
```

mdb_flush()

```
void mdb_flush(void);
```

刷新所有当前缓冲的输出。通常，`mdb` 的标准输出是按行缓冲的；只有遇到新行或者位于当前 dcmd 的结尾时，才会将使用 `mdb_printf()` 生成的输出刷新到终端（或其他标准输出目标）。但是，在某些情况下， 可能希望在列显新行之前显式刷新标准输出；可以使用 `mdb_flush()` 实现此目的。

mdb_nhconvert()

```
void mdb_nhconvert(void *dst, const void *src, size_t nbytes);
```

将存储在由 `src` 指定的地址的 `nbytes` 字节序列从网络字节顺序转换为主机字节顺序，并将结果存储在由 `dst` 指定的地址。 `src` 和 `dst` 参数可能是相同的，这种情况下将就地转换对象。 可以使用此函数从主机顺序转换为网络顺序，或从网络顺序转换为主机顺序，因为在任一情况下转换都是相同的。

mdb_dumpptr() 和 mdb_dump64()

```
int mdb_dumpptr(uintptr_t addr, size_t nbytes, uint_t flags,
                     mdb_dumpptr_cb_t func, void *data);
int mdb_dump64(uint64_t addr, uint64_t nbytes, uint_t flags,
                     mdb_dump64_cb_t func, void *data);
```

这些函数可以用于生成列显到标准输出的带格式十六进制和 ASCII 数据转储。 每个函数都接受 `addr` 参数（指定起始位置）、`nbytes` 参数（指定要显示的字节数）、如下所述的一组标志、func 回调函数（用于读取要显示的数据）和作为其最后一个参数传递到回调 func 的每个调用的数据参数。这些函数在各方面都是相同的，只不过 `mdb_dumpptr` 使用 `uintptr_t` 作为其地址参数，而 `mdb_dump64` 使用 `uint64_t`。 例如，将 `mdb_dump64` 与 `mdb_pread` 组合使用时，此不同是很有用的。 内置的 `::dump` dcmd 使用这些函数执行其数据显示。

flags 参数应该是以下一个或多个值的按位 OR：

`MDB_DUMP_RELATIVE`	相对于起始地址而不是每行的显式地址的数字行。
`MDB_DUMP_ALIGN`	在段落边界上对齐输出。
`MDB_DUMP_PEDANT`	显示全宽地址，而不是截断地址以适合 80 列输出。
`MDB_DUMP_ASCII`	在十六进制数据的旁边显示 ASCII 值。
`MDB_DUMP_HEADER`	显示有关数据的标题行。
`MDB_DUMP_TRIM`	仅读取并显示指定地址的内容，而不是读取和列显整行。
`MDB_DUMP_SQUISH`	通过在与前面行重复的行上放置 "*" 来取消重复行。
`MDB_DUMP_NEWDOT`	将点的值更新为超过函数读取的最后一个地址的地址。
`MDB_DUMP_ENDIAN`	按字节存储顺序调整。 此选项假定字大小等于由 `MDB_DUMP_GROUP()` 指定的当前组大小。此选项将始终关闭对齐方式、标题和 ASCII 显示，以避免输出冲突。 如果使用 `MDB_DUMP_ENDIAN` 设置了 `MDB_DUMP_TRIM`，则转储的字节数将向下舍入到最接近的字大小字节。
`MDB_DUMP_WIDTH(`*width*`)`	增加所显示的每行 16 字节段落数。 *width* 的缺省值为 1，最大值为 16。
`MDB_DUMP_GROUP(`*group*`)`	将字节组大小设置为 *group*。 缺省 *group* 大小为 4 字节。 *group* 大小必须是 2 的幂，且可以整除行宽。

mdb_one_bit()

```
const char *mdb_one_bit(int width, int bit, int on);
```

mdb_one_bit() 函数可以用于列显在其中打开或关闭相关单个位的位字段的图形化说明。对于创建与 snoop(1M) -v 的输出类似的位字段的详细显示，此函数是很有用的。 例如，以下源代码：

```
#define FLAG_BUSY       0x1



uint_t flags;



/* ... */



mdb_printf("%s = BUSY\n", mdb_one_bit(8, 0, flags & FLAG_BUSY));
```

生成以下输出：

```
.... ...1 = BUSY
```

位字段中的每个位都列显为一个句点 (.)，各个 4 位序列由空格分隔。相关位列显为 1 或 0，具体取决于 *on* 参数的设置。位字段的总 *width*（以位计）由 width 参数指定，相关位的位位置由 *bit* 参数指定。位从零开始编号。该函数将返回一个指针，该指针指向某个包含带格式位表示的大小适当、以空字符结尾的字符串。当前 dcmd 完成时，将自动对该字符串进行垃圾收集。

mdb_inval_bits()

```
const char *mdb_inval_bits(int width, int start, int stop);
```

可以将 mdb_inval_bits() 函数和 mdb_one_bit() 一起使用列显位字段的图形化说明。通过在适当的位位置上显示 'x'，此函数将位序列标记为无效或保留。位字段中的每个位都表示为一个句点 (.)，但由 start 和 stop 参数指定的位位置范围中的那些位除外。位从零开始编号。例如，以下源代码：

```
mdb_printf("%s = reserved\n", mdb_inval_bits(8, 7, 7));
```

生成以下输出：

```
x... .... = reserved
```

该函数将返回一个指针，该指针指向某个包含带格式位表示的大小适当、以空字符结尾的字符串。当前 dcmd 完成时，将自动对该字符串进行垃圾收集。

mdb_inc_indent() 和 mdb_dec_indent()

```
ulong_t mdb_inc_indent(ulong_t n);
```

```
ulong_t mdb_dec_indent(ulong_t n);
```

在列显输出行之前，这些函数递增和递减 MDB 将用空格自动缩进的列数。增量大小由列数 *n* 指定。每个函数都返回缩进以前的绝对值。尝试将缩进递减到低于零不起作用。在调用任一函数后，将相应地缩进对 `mdb_printf()` 的后续调用。如果 dcmd 完成或者它被用户强行终止，则调试器会将缩进自动恢复为其缺省设置。

mdb_eval()

```
int mdb_eval(const char *s);
```

评估并执行指定的命令字符串 *s*，就好像它由调试器从标准输入读取一样。如果成功，此函数返回 0；如果出错，此函数返回 -1。如果命令字符串包含语法错误，或者如果 `mdb_eval()` 执行的命令字符串被用户强行中止（使用页面调度程序或发出中断），则 `mdb_eval()` 失败。

mdb_set_dot() 和 mdb_get_dot()

```
void mdb_set_dot(uintmax_t dot);
```

```
uintmax_t mdb_get_dot(void);
```

设置或获取点（"." 变量）的当前值。模块开发者可能希望重新定位点，以便（例如）它引用 dcmd 读取的最后一个地址后面的地址。

mdb_get_pipe()

```
void mdb_get_pipe(mdb_pipe_t *p);
```

检索当前 dcmd 的管道输入缓冲区的内容。`mdb_get_pipe()` 函数供要占用完整的管道输入集但只执行一次的 dcmd 使用，以避免调试器针对每个管道输入元素重复调用 dcmd。在调用 `mdb_get_pipe()` 后，调试器不会再次在当前命令中调用 dcmd。例如，这可以用于构造对一组输入值进行排序的 dcmd。

管道内容放置在 dcmd 终止时对其进行垃圾收集的数组中，而数组指针存储在 `p->pipe_data` 中。 数组的长度放置在 `p->pipe_len` 中。如果未在管道的右侧执行 dcmd（即，未在其 `flags` 参数中设置 `DCMD_PIPE` 标志），则将 `p->pipe_data` 设置为 NULL，并将 `p->pipe_len` 设置为零。

mdb_set_pipe()

```
void mdb_set_pipe(const mdb_pipe_t *p);
```

将管道输出缓冲区设置为管道结构 *p* 说明的内容。管道值放置在数组 `p->pipe_data` 中，而数组的长度存储在 `p->pipe_len` 中。 调试器建立自己的此信息副本，因此调用方必须牢记在必要时释放 `p->pipe_data`。如果管道输出缓冲区以前不为空，则其内容将被替换为新数组。如果未在管道的左侧执行 dcmd（即，未在其 flags 参数中设置 `DCMD_PIPE_OUT` 标志），则此函数不起作用。

mdb_get_xdata()

```
ssize_t mdb_get_xdata(const char *name, void *buf, size_t nbytes);
```

将按名称指定的目标外部数据缓冲区的内容读入 *buf* 指定的缓冲区。*buf* 的大小由 *nbytes* 参数指定；复制到调用方缓冲区的字节数不会超过 *nbytes*。成功时将返回读取的总字节数；出错时将返回 -1。如果调用方要确定特定命名缓冲区的大小，则应该将 *buf* 指定为 NULL，并应该将 *nbytes* 指定为零。在这种情况下，`mdb_get_xdata()` 将返回缓冲区的总大小（以字节为单位），但不会读取任何数据。模块编写者可以通过外部数据缓冲区对无法通过模块 API 访问的目标数据进行访问。可以使用 `::xdata` 内置 dcmd 查看当前目标导出的命名缓冲区集。

其他函数

此外，模块编写者还可以使用以下 `string`(3C) 和 `bstring`(3C) 函数。它们保证具有与对应 Solaris 手册页中说明的函数相同的语义。

`strcat()`	`strcpy()`	`strncpy()`
`strchr()`	`strrchr()`	`strcmp()`
`strncmp()`	`strcasecmp()`	`strncasecmp()`
`strlen()`	`bcmp()`	`bcopy()`
`bzero()`	`bsearch()`	`qsort()`

附录 A

选项

本附录提供 MDB 命令行选项的参考。

命令行选项摘要

```
mdb [ -fkmuwyAFMS ] [ +o option ] [ -p pid ] [ -s distance]

     [ -I path ] [ -L path ] [ -P prompt ] [ -R root ]

     [ -V dis-version ] [ object [ core ] | core | suffix ]
```

支持以下选项：

-A
: 禁用 mdb 模块的自动装入功能。缺省情况下，mdb 尝试装入与用户进程或核心转储文件中的活动共享库相对应，或者与实时操作系统或操作系统崩溃转储中装入的内核模块相对应的调试器模块。

-F
: 强制接管指定的用户进程（如有必要）。缺省情况下，mdb 拒绝附加到已经受其他调试工具（如 truss(1)）控制的用户进程。但是通过使用 -F 选项，mdb 将附加到这些进程。这可能会在 mdb 和用于尝试控制进程的其他工具之间产生意外的交互。

-f
: 强制原始文件调试模式。缺省情况下，mdb 会尝试推断对象和核心转储文件操作数是引用用户可执行文件和核心转储，还是引用一对操作系统崩溃转储文件。如果无法推断出文件类型，则缺省情况下调试器将文件作为纯二进制数据进行检查。-f 选项用于强制 mdb 将参数解释为一组要检查的原始文件

-I
: 设置用于放置宏文件的缺省路径。使用 $< 或 $<< dcmd 来读取宏文件。路径是使用冒号 (:) 字符分隔的目录名称序列。-I include 路径和 -L library 路径（请参见下文）还可以包含以下任一标记：

  %i
  : 扩展为当前指令集体系结构 (ISA) 的名称（'sparc'、'sparcv9' 或 'i386'）。

%o　扩展为所修改路径的旧值。可用于将目录添加到现有路径的开头或末尾。

%p　扩展为当前平台字符串（uname -i 或存储在进程核心转储文件或崩溃转储中的平台字符串）。

%r　扩展为根目录的路径名。可以使用 -R 选项指定备用根目录。如果 -R 选项不存在，则根目录是从 mdb 可执行文件本身的路径动态派生的。例如，如果执行 /bin/mdb，则根目录为 /。 如果执行 /net/hostname/bin/mdb，则根目录可派生为 /net/hostname。

%t　扩展为当前目标的名称。这可以是文字字符串 'proc'（用户进程或用户进程核心转储文件）或 'kvm'（内核崩溃转储或实时操作系统）。

32 位 mdb 的缺省头文件路径为：

```
%r/usr/platform/%p/lib/adb:%r/usr/lib/adb
```

64 位 mdb 的缺省头文件路径为：

```
%r/usr/platform/%p/lib/adb/%i:%r/usr/lib/adb/%i
```

-k　强制内核调试模式。缺省情况下，mdb 会尝试推断对象和核心转储文件操作数是引用用户可执行文件和核心转储，还是引用一对操作系统崩溃转储文件。-k 选项用于强制 mdb 将这些文件假定为操作系统崩溃转储文件。如果未指定任何对象或核心操作数，但指定了 -k 选项，则 mdb 缺省将它们指定为 /dev/ksyms 目标文件和 /dev/kmem 核心转储文件。对 /dev/kmem 的访问仅限于 sys 组。

-K　装入 kmdb，停止实时运行的操作系统内核，然后转到 kmdb 调试器提示。此选项只应在系统控制台上使用，因为系统控制台上将显示后续的 kmdb 提示。

-L　设置用于放置调试器模块的缺省路径。模块是在启动时自动装入的或使用 ::load dcmd 装入的。路径是使用冒号 (:) 字符分隔的目录名称序列。-L 库路径还可以包含以上所示的 -I 的任一标记。

-m　禁用内核模块符号的按需装入功能。缺省情况下，mdb 会处理已装入内核模块的列表，然后执行按模块符号表的按需装入。如果指定了 -m 选项，则 mdb 不会尝试处理内核模块列表或提供逐个模块的符号表。因此，在启动时不会装入与活动内核模块相对应的 mdb 模块。

-M　预装入所有的内核模块符号。缺省情况下，mdb 会针对内核模块符号执行按需装入：如果地址为某模块的文本或者引用了数据部分，则读取该模块的完整符号表。通过 -M 选项，mdb 可在启动过程中装入所有内核模块的完整符号表。

-o *option*

启用指定的调试器选项。如果使用该选项的 +o 形式，则禁用指定的选项。除非下文另有说明，否则缺省情况下将禁用每个选项。mdb 可识别以下选项参数：

adb
实现更严格的 adb(1) 兼容性。提示符会设置为空字符串，并且许多 mdb 功能（如输出页面调度程序）将会禁用。

array_mem_limit=*limit*
设置 ::print 将显示的数组成员数的缺省限制。如果 *limit* 是特殊标记 none，则缺省情况下将显示所有数组成员。

array_str_limit=*limit*
设置 ::print 在列显 char 数组时将尝试显示为 ASCII 字符串的字符数的缺省限制。如果 *limit* 是特殊标记 none，则缺省情况下整个 char 数组都将显示为 string。

follow_exec_mode=mode
设置 exec(2) 系统调用之后的调试器行为。模式应该为以下命名常量之一：

ask	如果 stdout 是终端设备，则调试器在 exec(2) 系统调用返回之后将停止，然后提示用户决定是跟随 exec 还是停止。如果 stdout 不是终端设备，则 ask 模式将缺省为停止。
follow	调试器将通过自动继续执行目标进程并基于新的可执行文件重置其所有映射和符号表来跟随 exec。第 54 页中的 “与 exec 交互” 中将对跟随行为进行更详细的讨论。
stop	调试器在 exec 系统调用返回后将停止。第 54 页中的 “与 exec 交互” 中将对停止行为进行更详细的讨论。

follow_fork_mode=mode
设置 fork(2)、fork1(2) 或 vfork(2) 系统调用后的调试器行为。 模式应该为以下命名常量之一：

ask	如果 stdout 是终端设备，则调试器在 fork 系统调用返回之后将停止，然后提示用户决定是跟随父进程还是跟随子进程。 如果 stdout 不是终端设备，则 ask 模式将缺省为父进程。
parent	调试器将跟随父进程，并且会与子进程分离，同时将其设置为运行。
child	调试器将跟随子进程，并切会与父进程分离，同时将其设置为运行。

ignoreeof
  在终端上输入 EOF 序列 (^D) 时，调试器不会退出。如果要退出，必须使用 ::quit dcmd。

nostop
  如果指定了 -p 选项，或者应用了 ::attach 或 :A dcmd，则在附加到用户进程时请勿停止此进程。第 54 页中的 “进程的附加和释放” 中将对不停止行为进行更详细的介绍。

pager
  启用输出页面调度程序（缺省）。

repeatlast
  如果在终端上输入 NEWLINE 作为完整的命令，则 mdb 将使用当前点值重复执行前一个命令。-o adb 隐含此选项。

showlmid
  MDB 在使用除 LM_ID_BASE 和 LM_ID_LDSO 之外的链接映射的用户应用程序中，提供了对符号命名和标识的支持，如第 24 页中的 “符号名称解析” 中所述。除 LM_ID_BASE 或 LM_ID_LDSO 之外的链接映射上的符号将显示为 LMlmid‘library‘symbol，其中 lmid 是以缺省输出基数 (16) 表示的链接映射 ID。通过启用 showlmid 选项，用户可以选择配置 MDB，以显示所有符号和对象（包括与 LM_ID_BASE 和 LM_ID_LDSO 相关联的符号和对象）的链接映射 ID 的范围。用于处理目标文件名的内置 dcmd 将根据上面的 showlmid 值（包括 ::nm、::mappings、$m 和 ::objects）显示链接映射 ID。

-p *pid* 附加到指定的进程 id 并将其停止。mdb 使用 /proc/*pid*/object/a.out 文件作为可执行文件的路径名。

-P 设置命令提示符。缺省提示符为 '>'。

-R 设置路径扩展名的根目录。缺省情况下，根目录是从 mdb 可执行文件本身的路径名派生的。在路径名扩展过程中，会使用根目录替换 %r 标记。

-s *distance* 将用于地址到符号名转换的符号匹配距离设置为指定的 *distance*。缺省情况下，mdb 将距离设置为零，这样可启用智能匹配模式。每个 ELF 符号表项都包括值 V 和大小 S，表示函数或数据对象的大小（以字节为单位）。在智能模式下，如果 A 在范围 [V, V + S) 内，则 mdb 可将地址 A 与给定符号匹配。如果指定了任何非零距离，则会使用相同算法，但是给定表达式中的 S 始终为指定的绝对距离，并且会忽略符号大小。

-S 禁止处理用户的 ~/.mdbrc 文件。缺省情况下，如果用户的起始目录（如 $HOME 所定义）中存在宏文件 .mdbrc，则 mdb 会读取并处理该文件。如果存在 -S 选项，则不读取此文件。

-u 强制使用用户调试模式。缺省情况下，mdb 会尝试推断对象和核心转储文件操作数是引用用户可执行文件和核心转储，还是引用一对操作系统崩溃转储文件。-u 选项用于强制 mdb 假定这些文件不是操作系统崩溃转储文件。

-U
: 卸载 kmdb（如果已将其装入）。如果没有使用 kmdb 将内核调试器所使用的内存释放回可供操作系统使用的空闲内存，则应该将其卸载。

-V
: 设置反汇编程序版本。缺省情况下，mdb 会尝试推断适合于调试目标的反汇编程序版本。可以使用 -V 选项显式设置反汇编程序。::disasms dcmd 可列出可用的反汇编程序版本。

-w
: 打开指定的对象和核心转储文件以进行写入。

-y
: 针对 tty 模式发送显式终端初始化序列。某些终端需要显式初始化序列才能切换到 tty 模式。如果没有此初始化序列，则 mdb 可能无法使用终端功能（如 standout 模式）。

操作数

支持以下操作数：

object
: 指定要检查的 ELF 格式目标文件。通过 mdb 可以检查和编辑 ELF 格式可执行文件 (ET_EXEC)、ELF 动态库文件 (ET_DYN)、ELF 可重定位目标文件 (ET_REL) 和操作系统 unix.X 符号表文件。

core
: 指定 ELF 进程核心转储文件 (ET_CORE) 或操作系统崩溃转储 vmcore.X 文件。如果提供的 ELF 核心转储文件操作数没有对应的目标文件，则 mdb 将尝试使用几种不同的算法推断生成核心的可执行文件的名称。 如果未找到任何可执行文件，mdb 仍将执行，但是一些符号信息可能不可用。

suffix（后缀）
: 指定用于表示一对操作系统崩溃转储文件的数值后缀。例如，如果后缀为 '3'，则 mdb 会推断出应该检查文件 'unix.3' 和 'vmcore.3'。如果当前目录中存在同名的实际文件，则不会将数字串解释为后缀。

退出状态

将返回以下退出值：

0
: 调试器已成功完成执行。

1
: 发生了致命错误。

2
: 指定的命令行选项无效。

环境变量

支持以下环境变量：

HISTSIZE 此变量用于确定命令历史记录列表的最大长度。如果不存在此变量，则缺省长度为 128。

HOME 此变量用于确定用户起始目录（可能驻留有 `.mdbrc` 文件）的路径名。如果不存在此变量，则不会进行任何 `.mdbrc` 处理。

SHELL 此变量用于确定 shell（用于处理使用 ! 元字符请求的 shell 转义）的路径名。如果不存在此变量，则会使用 `/bin/sh`。



附录 B

注意事项

警告

以下警告信息适用于使用 MDB 的情况。

使用错误恢复机制

调试器及其 dmod 在同一地址空间中执行，因此错误的 dmod 很可能会导致 MDB 转储核心或行为异常。第 33 页中的 “信号处理”中介绍的 MDB `resume` 功能针对这些情况提供了一种受限制的恢复机制。但是，MDB 无法明确知道相关的 dmod 是仅损坏了自己的状态还是损坏了调试器的全局状态。因此，无法保证 `resume` 操作是安全的或可防止调试器将来崩溃。执行 `resume` 之后最安全的做法保存所有重要的调试信息，然后退出并重新启动调试器。

使用调试器修改实时操作系统

使用调试器修改（即写入）实时运行的操作系统的地址空间是极其危险的，如果用户损坏了内核数据结构，可能会导致系统崩溃。

使用 `kmdb` 停止实时操作系统

使用 `kmdb` 停止实时操作系统（使用 `mdb -K` 或通过在实时操作系统中设置断点）适用于开发者，而不适用于生产系统。如果通过 `kmdb` 停止操作系统内核，则操作系统服务和联网将不会执行，并且网络中与目标系统相关的其他系统将无法访问目标系统。

注意事项

进程核心转储文件的检查限制

对于 Solaris 2.6 之前的 Solaris 操作系统发行版生成的进程核心转储文件，MDB 不会为对其进行检查提供支持。如果在一种操作系统发行版上检查其他操作系统发行版中的核心转储文件，则运行时链接编辑器调试接口 (`librtld_db`) 可能无法初始化。在这种情况下，共享库的符号信息将不可用。 此外，由于用户核心转储文件中不存在共享映射，因此共享库的文本部分和只读数据可能与转储核心时进程中存在的数据不匹配。不能在 Solaris SPARC 系统上检查 Solaris Intel 系统中的核心转储文件，反之亦然。

崩溃转储文件的检查限制

Solaris 7 和早期发行版中的崩溃转储只能借助对应操作系统发行版中的 `libkvm` 进行检查。如果一种操作系统发行版中的崩溃转储是使用其他操作系统发行版中的 dmod 检查的，则内核实现中的更改可能会使一些 dcmd 或 walker 无法正常工作。如果 MDB 检测到此情况，将会发出一条警告消息。不能在 Solaris SPARC 系统上检查 Solaris Intel 系统中的崩溃转储，反之亦然。

32 位调试器和 64 位调试器之间的关系

MDB 支持对 32 位程序和 64 位程序进行调试。MDB 在检查目标并确定其数据模型后，将自动重新执行具有与目标相同的数据模型的 mdb 二进制代码（如有必要）。此方法简化了编写调试器模块的任务，因为装入的模块将使用与主要目标相同的数据模型。只能使用 64 位调试器调试 64 位目标程序。64 位调试器只能在运行 64 位操作环境的系统上使用。

`kmdb` 可用内存的限制

可供 `kmdb` 使用的内存是在装入调试器时分配的，在此之后无法进行扩展。如果调试器命令尝试分配的内存多于可用内存，这些命令将无法执行。调试器会尝试从内存不足的情况正常恢复，但是在紧急情况下可能会强制终止系统。在使用 32位操作系统内核的 x86 平台上，系统内存约束尤其苛刻。

开发者信息

`mdb`(1) 手册页详细介绍了内置 `mdb` 功能，以便于开发者参考。头文件 `<sys/mdb_modapi.h>` 包含 MDB 模块 API 中函数的原型，`SUNWmdbdm` 软件包用于为目录 `/usr/demo/mdb` 中的示例模块提供源代码。

附录 C

从 adb 和 kadb 转换

从使用传统的 adb(1) 实用程序到使用 mdb(1) 的转换相对来说是简单的：MDB 为 adb 语法、内置命令和命令行选项提供了演化的兼容性。MDB 尝试为所有的现有 adb(1) 功能提供兼容性，但是它与 adb(1) 并不是逐错误兼容。本附录简要讨论 mdb(1) 没有精确模仿的几项 adb(1) 功能，以便引导用户使用新功能

命令行选项

MDB 提供 adb(1) 所识别的命令行选项的超集。所有的 adb(1) 选项都是受支持的，且具有与以前相同的含义。/usr/bin/adb 路径名作为调用 mdb(1) 的链接提供，它自动启用增强的 adb(1) 兼容模式。执行 /usr/bin/adb 链接等效于执行带 -o adb 选项的 mdb 或者在启动调试器后执行 ::set -o adb。

语法

MDB 语言遵循与 adb(1) 语言相同的语法，以便为传统的宏和脚本文件提供兼容性。新增的 MDB dcmd 使用扩展形式 ::name，以便将它们与以 : 或 $ 为前缀的传统命令区分开。也可以在 dcmd 名称的右侧计算表达式，方法是将它们括在前面是美元符号的方括号 ($[ ]) 中。与 adb(1) 类似，以叹号 (!) 开头的输入行指示该命令行应该由用户的 shell 执行。在 MDB 中，调试器命令也可能以叹号为后缀，以指示应该将其输出传输到叹号后面的 shell 命令。

在 adb(1) 中，二元运算符是左关联的，其优先级低于一元运算符。在输入行上，二元运算符是严格按照从左向右的顺序计算的。在 MDB 中，二元运算符是左关联的，其优先级低于一元运算符，但是二元运算符按照第 22 页中的 “二元运算符”中的表所示的优先顺序运算。运算符符合 ANSI C 中的优先顺序。可能需要更新不显式括起二义性表达式的传统 adb(1) 宏文件以用于 MDB。例如，在 adb 中，以下命令的计算结果是整数值 9：

```
$ echo "4-1*3=X" | adb

        9
```

在 MDB 中，与在 ANSI C 中一样，运算符 "*" 的优先级高于 "-"，因此结果是整数值 1：

```
$ echo "4-1*3=X" | mdb

        1
```

观察点长度说明符

MDB 识别的观察点长度说明符语法与 adb(1) 中所述的语法不同。特别是，adb 观察点命令 :w、:a 和 :p 允许在冒号和命令字符之间插入以字节为单位的整数长度。在 MDB 中，应该在初始地址后面将计数指定为重复计数。简单地说，以下 adb(1) 命令：

```
123:456w

123:456a

123:456p
```

在 MDB 中指定为

```
123,456:w

123,456:a

123,456:p
```

MDB ::wp dcmd 提供用于创建用户进程观察点的更全面的工具。类似地，不支持传统的 kadb 长度修饰符命令 $l。因此，应该为 kmdb 中使用的每个 ::wp 命令指定观察点大小。

地址映射修饰符

MDB 中不存在用于修改虚拟地址映射和目标文件映射的段的 adb(1) 命令。具体而言，MDB 既不识别也不支持 /m、/*m、?m 和 ?*m 格式说明符。这些说明符过去用于手动修改当前对象和核心转储文件的有效可寻址范围。-{}- MDB 自动地正确识别此类文件的可寻址范围，并在调试实时进程时更新范围，因此不再需要这些命令。

输出

一些命令的精确文本输出格式在 MDB 中是不同的。使用相同的基本规则格式化宏文件，但是可能需要更改依赖于某些命令的逐字符精确输出的 shell 脚本。具有 shell 脚本（用于解析 `adb` 命令的输出）的用户在转换到 MDB 时，将需要重新验证和更新这样的脚本。

延迟断点

传统的 `kadb` 实用程序支持延迟断点语法，该语法与现有的 `adb` 语法不兼容。这些延迟断点的指定方法是：在 `kadb` 中使用语法 *module#symbol*`:b`。要在 `kmdb` 中设置延迟断点，请使用 MDB `::bp` dcmd，如第 6 章中所述。

x86: I/O 端口访问

传统的 `kadb` 实用程序使用 `:i` 和 `:o` 命令提供对 x86 系统上 I/O 端口的访问。在 `mdb` 或 `kmdb` 中不支持这些命令。对 x86 系统上 I/O 端口的访问由 `::in` 和 `::out` 命令提供。



附录 D

从 crash 转换

从使用传统的 crash 实用程序到使用 mdb(1) 的转换相对来说是简单的：MDB 提供了大多数“固定的”崩溃命令。通过 MDB 的其他可扩展性和交互功能，程序员可以了解当前命令集未检查的系统各方面。本附录简要讨论了 crash 的几项功能，并提供了指向等效 MDB 功能的指针。

命令行选项

mdb 不支持 crash -d、-n 和 -w 命令行选项。crash 转储文件和名称列表（符号表文件）按名称列表、崩溃转储文件的顺序指定为 mdb 的参数。 要检查实时内核，应该指定不带其他参数的 mdb -k 选项。 如果要将 mdb 的输出重定向到文件或其他输出目标，应该在命令行上使用 mdb 调用后跟相应的 shell 重定向运算符，或者使用 ::log 内置 dcmd。

MDB 中的输入

通常，MDB 中的输入与 crash 类似，但函数名称（在 MDB 中即为 dcmd 名称）的前缀为 "::"。 一些 MDB dcmd 接受位于 dcmd 名称前面的前导表达式参数。 与 crash 一样，字符串选项可以跟随 dcmd 名称。 如果函数调用后有 ! 字符，则 MDB 还将创建一个到指定 shell 管道的管道。 缺省情况下，在 MDB 中指定的所有即时值都是按十六进制解释的。即时值的基数说明符在 crash 和 MDB 中不同，如表 D–1 所示：

表 D–1 基数说明符

crash	mdb	基数
0x	0x	十六进制（以 16 为基数）
0d	0t	十进制（以 10 为基数）
0b	0i	二进制（以 2 为基数）

许多 crash 命令接受以槽号或槽范围作为输入参数。Solaris 操作系统不再是按照槽构建的，因此 MDB dcmd 未提供对槽号处理的支持。

函数

crash 函数	mdb dcmd	注释
?	::dcmds	列出可用的函数。
!command	!command	转义到 shell 并执行命令。
base	=	在 mdb 中，= 格式字符可以用于将左侧的表达式值转换为任何已知格式。提供了八进制、十进制和十六进制格式。
callout	::callout	列显调用表。
class	::class	列显调度类。
cpu	::cpuinfo	列显有关在系统 CPU 上分发的线程的信息。如果需要特定 CPU 结构的内容，则用户应该将 $<cpu 宏应用于 mdb 中的 CPU 地址。
help	::help	列显指定 dcmd 的说明或常规帮助信息。
kfp	::regs	mdb ::regs dcmd 显示完整的内核寄存器集，包括当前的栈帧指针。$C dcmd 可以用于显示栈反向跟踪（包括帧指针）。
kmalog	::kmalog	显示内核内存分配器事务日志中的事件。
kmastat	::kmastat	列显内核内存分配器事务日志。
kmausers	::kmausers	列显有关具有当前内存分配的内核内存分配器中等用户和大用户的信息。
mount	::fsinfo	列显有关已挂载文件系统的信息。
nm	::nm	列显符号类型和值信息。
od	::dump	列显给定区域的带格式内存转储。在 mdb 中，::dump 以 ASCII 和十六进制混合显示区域的信息。
proc	::ps	列显活动进程表。
quit	::quit	退出调试器。
rd	::dump	列显给定区域的带格式内存转储。在 mdb 中，::dump 以 ASCII 和十六进制混合显示区域的信息。
redirect	::log	在 mdb 中，可以使用 ::log 以全局方式将输入和输出的输出重定向到日志文件。

crash 函数	mdb dcmd	注释
search	::kgrep	在 mdb 中，::kgrep dcmd 可以用于在内核的地址空间中搜索特定值。模式匹配内置 dcmd 也可以用于在物理、虚拟或目标文件地址空间中搜索模式。
stack	::stack	可以使用 ::stack 获取当前栈跟踪。 可以使用 ::findstack dcmd 确定特定内核线程的栈跟踪。 可以使用 / 或 ::dump dcmd 以及当前的栈指针获取当前栈的内存转储。可以将 `$<stackregs` 宏应用于栈指针，以获取按帧保存的寄存器值。
status	::status	显示有关调试器正在检查的系统或转储的状态信息。
stream	::stream	mdb ::stream dcmd 可以用于设置特定内核 STREAM 的结构的格式和显示该结构。如果需要活动的 STREAM 结构的列表，则用户应该在 mdb 中执行 ::walk stream_head_cache，并将生成的地址传输到相应的格式化 dcmd 或宏。
strstat	::kmastat	::kmastat dcmd 显示 `strstat` 函数所报告的信息的超集。
trace	::stack	可以使用 ::stack 获取当前栈跟踪。可以使用 ::findstack dcmd 确定特定内核线程的栈跟踪。可以使用 / 或 ::dump dcmd 以及当前的栈指针获取当前栈的内存转储。可以将 `$<stackregs` 宏应用于栈指针，以获取按帧保存的寄存器值。
var	$<v	列显全局 `var` 结构中的可调系统参数。
vfs	::fsinfo	列显有关已挂载文件系统的信息。
vtop	::vtop	列显给定虚拟地址的物理地址转换。

索引

F

I

K

M

R

S

U

W

变

标

错

符

格

管

宏

回

加

进

扩

联

命

目

内

配

平

任

设

事

输

搜

算

同

未

文

字